Analysis of Clinical and Animal Laboratory Medical Charts Using Excel and R

US Army Institute of Surgical Research Technical Report Number 2019-1

William L Baker

June 19, 2019

DEPARTMENT OF THE ARMY
US ARMY INSTITUTE OF SURGICAL RESEARCH
3698 CHAMBERS PASS, STE B
JBSA FORT SAM HOUSTON, TEXAS 78234-7767

Acknowledgements

The author wishes to thank Dr. Jose Salinas, Dr. A. Batchinsky and Dr. L. Cancio as well as the patient members of the MOST Task Area for the opportunity to participate in their numerous studies.

Disclaimer

Abstract

The standard of care for medical data documentation in the ICU involves the recording of hourly vitals. These vitals are used for a quick summary of patient status and for visualizing trends as part of routine medical care. In a similar way, large animal research models of critical care use hourly vitals recordings to track subject state and treatment response. In addition to the hourly vitals, time points of interest to research may be recorded as frequently as every five minutes. These recordings are entered into a series of Excel spreadsheets for convenience of data entry and editing as well as easy generation of trend graphs. For each animal, a single Excel workbook contains sheets for hourly vitals, lab values, and demographics as well as study-specific devices, procedures, and results. In the analysis phase, it is necessary to pull each subject's data from the series of workbooks into a single coherent dataset from which time points can be correlated and groups can be compared both graphically and numerically. This document gives a procedure for assembling the Excel data into a single dataset for analysis using R.

Contents

Introduction

In this paper we present a method for extracting data from two diverse datasets for either combined or separate analysis. The first dataset comes from a large number of laboratory medical charts which are stored as Excel spreadsheets. These medical charts resemble the hourly vitals charts of any hospital, with the exception that the time-points are not explicitly entered hourly. For the purpose of the laboratory, the vitals are entered at experiment time-points, which are typically hourly if not more frequent.

The second dataset comes directly from the medical instruments. It is collected continuously from the most significant medical devices without requiring operator intervention or input. Most of this data capture is digital, eliminating the need for analog to digital conversion or analog signal calibration.

Both datasets are indispensable in their own way. The medical chart data is entered and verified by humans. Although humans are susceptible to recording of spurious events that are not indicative of trends as well as transcription errors, humans will often double-check abnormal or unlikely values and proceed to fix problems with fluid lines and instruments. Humans will also enter data that is not available for automated recording, such as the time of an experiment event. On the other hand, although the machine recorded data is unverified, it is very good for visualizing trends and finding errors in the human data. The datasets complement each other.

In most laboratories, Excel is the first tool of choice for data collection and analysis. Its strengths and weaknesses are well known. Certainly for very large datasets as well as datasets that span multiple Excel workbooks, this tool is inadequate. We have both of these cases. Our solution is to use a well-known scripting language, R, to extract the data from the Excel workbook and carry the analysis from raw data to end product – which is the essence of reproducible analysis. This method allows us to document, inspect, and verify every step of the process, and this is the foundation of reproducible research.

For those who have not taken a statistics course recently and have not interacted with recent graduates, you may be unfamiliar with R. This quirky statistics-centric programming language has been integrated into most university statistics courses. R is free, state-of-the-art, well documented and used daily by professional statisticians. It is a valuable resource in its own right and will amply reward the research who learns it. No previous knowledge of the R language is needed to follow this tutorial, but it may be necessary to consult online resources for a complete understanding. Sources of additional training material can be found for free in online courses at Coursera:

https://www.coursera.org/learn/data-management

https://www.coursera.org/learn/reproducible-research

R: The Analysis and Statistical Language

"Statistics is the grammar of science." Karl Pearson

Do I need to learn a programming language? The answer is, "No, unless you care about your data analysis". But, if you do not care about your data, this guide is not for you. Or, if you are happy to rely on other members of your team to do high quality reproducible data analysis, then this guide is not for you.

The purpose of this guide is to enable the average researcher at ISR to perform basic reproducible data analysis of vitals from data found in Excel medical charts and ASCII vitals recordings through the use of R. R is a language for statistics together with an integrated suite of software facilities for data manipulation, calculation and graphical display. The intent of this technical manual is to be self-sufficient, but true understanding of R and its environment will require additional resources. Additional reference material can be found for free in online:

https://cran.r-project.org/doc/manuals/R-intro.html

Prerequisites: R and R Studio

You will need to install R and optionally R Studio to follow this guide. R can be obtained from the R project web site: https://www.r-project.org. For Windows, R comes with a minimal interactive GUI program called Rgui. This base install is functional but not intuitive. Running Rgui will open a blank document for entering commands.

R Studio builds on top of the basic R package and provides a friendlier environment for the user. R Studio is not as intimidating as Rgui, and a number of YouTube tutorials will enable a new user to feel comfortable in this environment in under an hour. For this tutorial, either of approach will work.

DoD networks require a Certificate of Networthiness (CoN). Both the DIACAP and the more recent Risk Management Framework require or acknowledge the usage of the Certificate of Networthiness. Both R 3.x and R Studio 1.x have active CoN's, a copy of which has been included in the appendix, *DoD Security and Certificates of Networthiness*. See the CoN for security measures that must be taken to enable use of these products.

Many of the scripts below require an optional package to be installed: "ggplot2". This package and others can be included during the R installation or by following the instructions when those packages are needed. Installation instructions for "ggplot2" can be found below in the section *Libraries and More Commands*.

The Interactive Prompt and Finding Help

R is used both interactively and through scripts. We will focus on interactively usage of R through the command prompt. If you have not used the R command prompt previously, please refer to one of the above references before continuing. At a minimum, one should understand how to access the online help system for additional information on packages and commands, recognize the assignment operator "<-", and recognize the pound sign "#" as the beginning of a comment.

All R packages are required to have documentation in the online help system as well as PDF formatted documents. Some packages have additional documentation in vignettes. To find help on using a particular command, enter "?" followed by the command of interest. It is also possible to search the help system for references to a top by entering "??" followed by a topic name. See the examples below:

```
> ? read.tables                   # this will search for the documentation of the read.tables() command
> ?? read.tables                  # this will search all documentation for references to read.tables
> browseVignettes()               # list the available vignette documentation
```

Vectors, the Building Blocks

The simplest data structure in R is the numeric *vector*, which is a single entity consisting of an ordered collection of numbers. This should seem peculiar to programmers in any other language. What about a simpler type, like a single integer? Take a look. In R we can enter a simple calculation such as: 1 + 2. What is the result? A vector of length 1. Try it:

```
> 1 + 2
[1] 3
```

It certainly looks like an integer. The value is 3. But what about the "[1]" prefix in the output? Oh yes, it is, the first element of an array of length one. Being vector or array centric, it should be easy to create an array of several numbers, and indeed it is! Enter the following at the > prompt:

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
>
```

Nothing was printed except an empty prompt. It appears we created an array of length 5, consisting of various numbers. Did we? Enter "x" at the next prompt to see the value.

```
> x
[1] 10.4  5.6  3.1  6.4 21.7
```

Maybe we have an array of five values. We still have this "[1]" prefix, but there is no "[2]", "[3]",… This would make sense if only the first number on each line were numbered. Let's test that hypothesis by creating an array from 1 to 50. R has a short-hand method to do this:

```
> x <- c(1:50)
> x
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50
>
```

Indeed! The expected result. What happens if we try a simple case of adding 1 to x? Try and see.

```
> x <- x + 1
> x
  [1]   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
 [19]  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37
 [37]  38  39  40  41  42  43  44  45  46  47  48  49  50  51
>
```

For programmers in other languages, this result is unexpected. "1" was added to every element of x without the need for an explicit loop. Being an array, it should be possible to assign a value to individual elements of the array with familiar [] syntax used in most programming languages, which is done as follows:

```
> x[1] <- 2017
> x
 [1] 2017    3    4    5    6    7    8    9   10   11   12   13   14   15   16
[16]   17   18   19   20   21   22   23   24   25   26   27   28   29   30   31
[31]   32   33   34   35   36   37   38   39   40   41   42   43   44   45   46
[46]   47   48   49   50   51
>
```

There are more surprises along the way. In addition to arrays, the most important data structure in R is the data frame. Think of a data frame as a spreadsheet: it has rows and columns. Each column represents a single variable, and each row holds a single record. But before we get to datasets, let's introduce some basic commands and a few essential data types.

Basic Commands

The table below contains the most basic commands that will be used throughout this document.

Command	**Description**
	Help System
#	The beginning of a comment, generally used for code documentation.
? abc	Help about the command "abc".
?? abc	Search the online help system for any references to "abc"
	Variable Management
class(x)	Return the class of variable "x". The variable class will be something like "integer", "numeric", "character", or "POSIXct". There are hundreds of possible classes.
head(x)	Print the first few elements of "x"
length(x)	Print the number of elements in array "x".
ls()	List the defined variables in the current workspace environment.
rm(x)	Delete variable "x" from memory.
unclass(x)	Expose fundamental storage elements of a complex class
	Basic Statistics
max(x)	Return the maximum value in array "x".
mean(x)	Return the mean value of array "x".
min(x)	Return the minimum value in array "x".
sum(x)	Return the sum of all values in array "x".
summary(x)	Print a set of information related to variable "x"
	String Functions
cat(x, "b", "c", "\n")	Print the value of variable "x", followed by the literal letters "b", "c" and a new line.

paste(x, y, z)	Return the values of variables x, y, and z joined together with a space between each value.

	Date Functions
as.POSIXct()	Convert, possibly parse, a value to the POSIXct data structure.
as.POSIXlt()	Convert, possibly parse, a value to the POSIXct data structure.
as.difftime(value, units=)	Creates an object that represents the difference of two date/time objects with an attribute indicating the units
strptime(value, format=)	Parse character representations of date and time objects.
Sys.time()	Return the current date and time.
	File Management
dir.create(path)	Creates the last element of the path, unless the optional argument `recursive = TRUE`. Trailing path separators are discarded. On Windows, drives are allowed in the path specification and unless the path is rooted, it will be interpreted relative to the current directory on that drive.
getwd()	returns an absolute filepath representing the current working directory of the **R** process; `setwd(dir)` is used to set the working directory to `dir`
setwd(dir)	Set the working directory to the "dir" indicated. `setwd` uses the same conventions as `getwd`.
list.files(path)	Produce a character vector of the names of files or directories in the named directory or current working directory if path is not specified.
source(file)	Read from the named file or URL or connection or expressions directly. Input is read and `parse`d from that file until the end of the file is reached, and then the parsed expressions are evaluated sequentially in the chosen environment.
	Data Access
read.csv(file)	Reads a file in CSV format and creates a data frame.
read.table(file)	Reads a file in table format and creates a data frame.
write.csv(x, file)	Write the R object "x" to the file specified using the CSV format.

Dates and Times

There are multiple ways to represent dates and times in R. The most common way to obtain a date is to parse a string representation. The data type produced depends on the function used to parse the string. The most commonly used parsing routines are as.POSIXct() and as.POSIXlt(). POSIXct internally represents a data as an integer, which is the number of seconds since Jan 1, 1970. The date range of POSIXct is therefore restricted to a value between 1902 and 2037.

```
# POSIXct example

> today <- Sys.Date()                # Date, without the time
> today_w_time <- Sys.time()         # Date and time
> class(today_w_time)                # What type of variable is this?
[1] "POSIXct" "POSIXt"
> today_w_time                       # show the string representation
"2017-09-10 13:30:40 CDT"
> as.numeric( today_w_time )         # Number of seconds since jan 1, 1970
1505068319
> as.POSIXct(as.numeric(Sys.time()), origin="1970-01-01")
"2017-09-10 13:30:40 CDT"
```

Parsing dates from strings can be done with the POSIX functions as well as the strptime function. All parsing functions take a format string which is described in the online documentation. It is easy to make mistakes with dates. For instance, the hour format %H expects a value 0-23 and ignores the AM/PM indication. The format specifier %I expects a value 0-11 indicating the hour and respects the AM/PM indication.

```
# strptime example

> fdate <- "12/31/2017 08:00:00 PM"
> pdate <- strptime(fdate, "%m/%d/%Y %I:%M:%S %p", tz="America/Chicago" )
> pdate
[1] "2017-12-31 20:00:00 CST"

# Incorrect use of %H, and probably not the result you want.

> pdate <- strptime(fdate, "%m/%d/%Y %H:%M:%S %p", tz="America/Chicago" )
> pdate
[1] "2017-12-31 08:00:00 CST"
```

Another class that represents a date and time is POSIXlt. POSIXlt internally represents a data as a series of values: integers for the year, month, day, hour, etc. POSIXlt also requires a time zone. POSIXlt is therefore capable of almost any date.

```
# POSIXlt example

> tm -> as.POSIXlt( Sys.time() )          # Get today's date
> tm                                      # A simple printout
"2017-09-10 13:30:40 CDT"
> unclass( tm )                           # Show storage info in detail
$sec
[1] 54.00688                              # POSIXlt can handle fractions of a second
$min
[1] 53
$hour
[1] 13
$mday                                     # day of month: 1-31
[1] 10
$mon                                      # month: 0-11
[1] 8
$year                                     # years since 1900
[1] 117
$wday
[1] 0
$yday
[1] 252
$isdst                                    # it understands daylight savings time
[1] 1
$zone
[1] "CDT"
$gmtoff
[1] -18000
attr(,"tzone")                            # time zone
[1] ""    "CST" "CDT"
>
```

Time deltas are a unique data type, called "difftime". Dates can be manipulated by adding and subtracting difftime objects.

```
# difftime example

> z <- as.difftime( 7, units = "days")
> tm <- Sys.time()
> tm – z
[1] "2017-09-03 14:00:27 CDT"
```

Excel uses a single decimal number to represent a date, with the value 1.0 being 24 hours, or 1 day. Likewise, 0.5 would be half a day, or 12 hours. You can see this in Excel by formatting an Excel date column to a numeric type. The Excel date is simply this decimal number defined and formatted as a human readable calendar date. Conversion to an R date is often simplest by reading the Excel column as a formatted string, then parsing the string to give a compatible R date.

Lists and Factors

Lists and Factors are the last two important types of R data, other than dataframes. Lists are easy to understand as arrays indexed by words instead of numbers. A little care must be taken with the single brackets, [and], as opposed to the double brackets, [[and]]. Numbers are always enclosed in single brackets, while words are always enclosed in double brackets.

Here is sample usage:

```
t <- list()
> t[["date"]] <- Sys.time()
> t[["animal_no"]] <- 8250
> t[1]
$date
[1] "2017-09-11 19:16:49 CDT"

> t[2]
$animal_no
[1] 8250

> t[["date"]]
[1] "2017-09-11 19:16:49 CDT"
>
```

Factors too are usually simple. Time points are a good example of a usage case for factors. Let's consider an experiment with three time points: "Baseline", "Injury", and "EOS". In this example, these are the only possible values for a time point. To assign the value "IL2" to a time point would be an error. It may also be important to assign factors a specific order, particularly in the case of time points. Factors were made for this, and for some purposes such as categorical separation of plots, factors are even required. But R is not perfect, and it tends to decide too often that a variable is a factor. This happens most often when reading data from text files. It is often best to specify the column type when the data is read as part of the read.csv() or read.table() command.

```
> tp <- as.factor( c("Baseline", "Injury", "EOS" ) )
> tp
[1] Baseline Injury   EOS
Levels: Baseline EOS Injury

# the following assignment succeeds
> tp[4] <- "Baseline"

# But, since "BaseLine" is not a factor, the following assignment fails.
> tp[4] <- "BaseLine"
Warning message:
In `[<-.factor`(`*tmp*`, 4, value = "BaseLine") :
  invalid factor level, NA generated

# Inspecting the resulting tp array, we see that NA has been inserted in position 4
> tp
[1] Baseline Injury   EOS      <NA>
Levels: Baseline EOS Injury
```

Data Frames

A data frame is a fundamental type in R, and it is used everywhere. Without an understanding of data frames, you won't get far. A data frame is essentially the same as a simple spreadsheet in Excel. It has rows and columns. Typically, a column represents a single variable, such as time, and each row is a timepoint. Both the columns and rows are numbered, and the columns are named as well.

Data frames have some constraints. First, all columns must have the same number of rows. Second, each column can have only one type of data. As an example, we can create a data frame that has three hourly readings for systolic, diastolic, and MAP. One way to do this is to create the data columns, then combine them to produce into a data frame.

```
# Basic dataframe example

> time <- c("1am", "2am", "3am")
> sys <- c(120, 110, 120)
> dia <- c( 60, 60, 70)
> map <- c( 85, 80, 95 )
>
> ds <- data.frame( time, sys, dia, map )
> ds
  time sys dia map
1  1am 120  60  85
2  2am 110  60  80
3  3am 120  70  95
>
```

Beautiful.

Given a dataframe, we can inspect it in various ways. Generally, we are interested in knowing how many columns it has, the column names, and the number of rows. See the example below.

```
# Inspecting the dataframe example

> class(ds)
[1] "data.frame"
> names(ds)
[1] "time" "sys"  "dia"  "map"
> ncol(ds)
[1] 4
> nrow(ds)
[1] 3
>
```

The rows, columns, and individual values of a dataframe can accessed directly by name or by index in much the same way as a two dimensional array. Some examples are shown below.

```
> #
> # Examples of access to elements of a dataframe
> #

> # Access column by name
> ds$time
[1] 1am 2am 3am
Levels: 1am 2am 3am

> # Access column by index
> ds[[1]]
[1] 1am 2am 3am
Levels: 1am 2am 3am

> # Access a single value by row and column
> ds[1,1]
[1] 1am
Levels: 1am 2am 3am

> # Access of an entire row
> ds[1,]
  time sys dia map
1  1am 120  60  85

> # Access of an entire column
> ds[,1]
[1] 1am 2am 3am
Levels: 1am 2am 3am
```

Libraries and More Commands

Samson, where does your strength come from?

Delilah, it comes from my package manager, core packages, and 1000's of contributors across the world!

The strength of R comes from its 1000's of contributors. When you obtain R initially, your only installed the core packages, also called the "base" packages. Each package contains additional commands, code, and documentation to perform a specific task or group of tasks.

We will need more packages that are not included in the base set. One such packages is "ggplot2". Most R packages are in well-known repositories on the internet. There are approximately 10 well-known repositories, although even fewer repositories are usually needed.

To browse the available packages and select specific packages to install through the GUI, enter the command *install.packages()*. This will present the user with a list of source locations from which to obtain the packages. Selecting the location will result in another popup showing the packages that are available from the selected location. *(Figure 1: CRAN Mirrors and Packages)* All locations should have the same packages and package versions. The list is extensive.

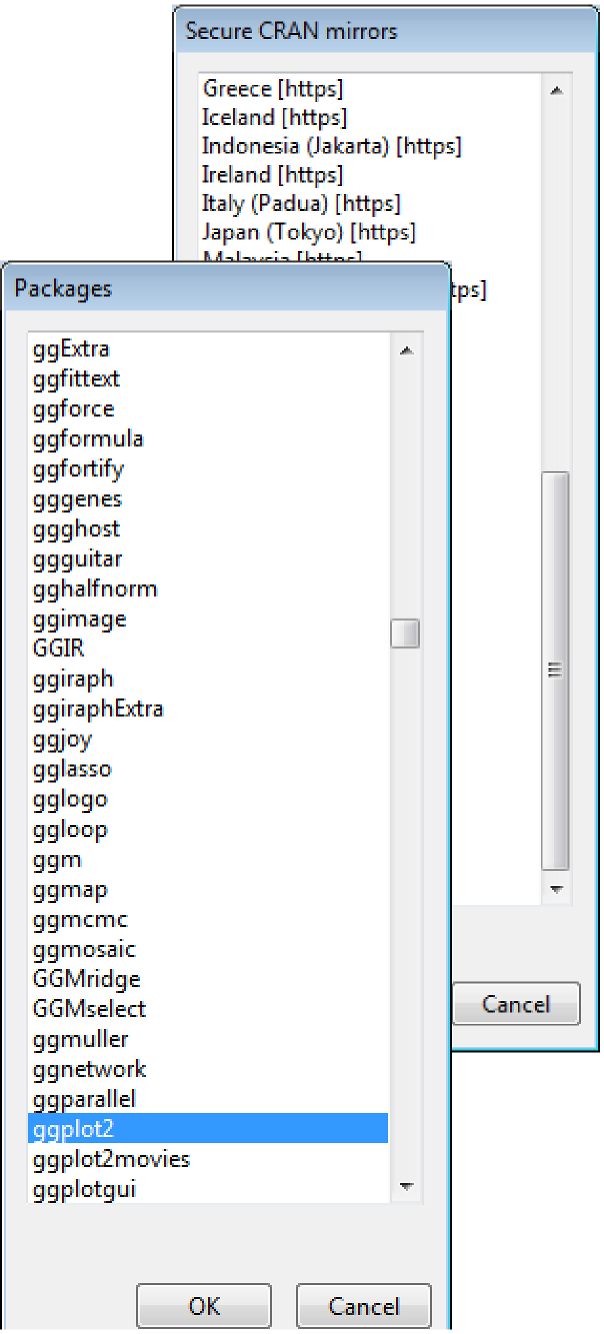
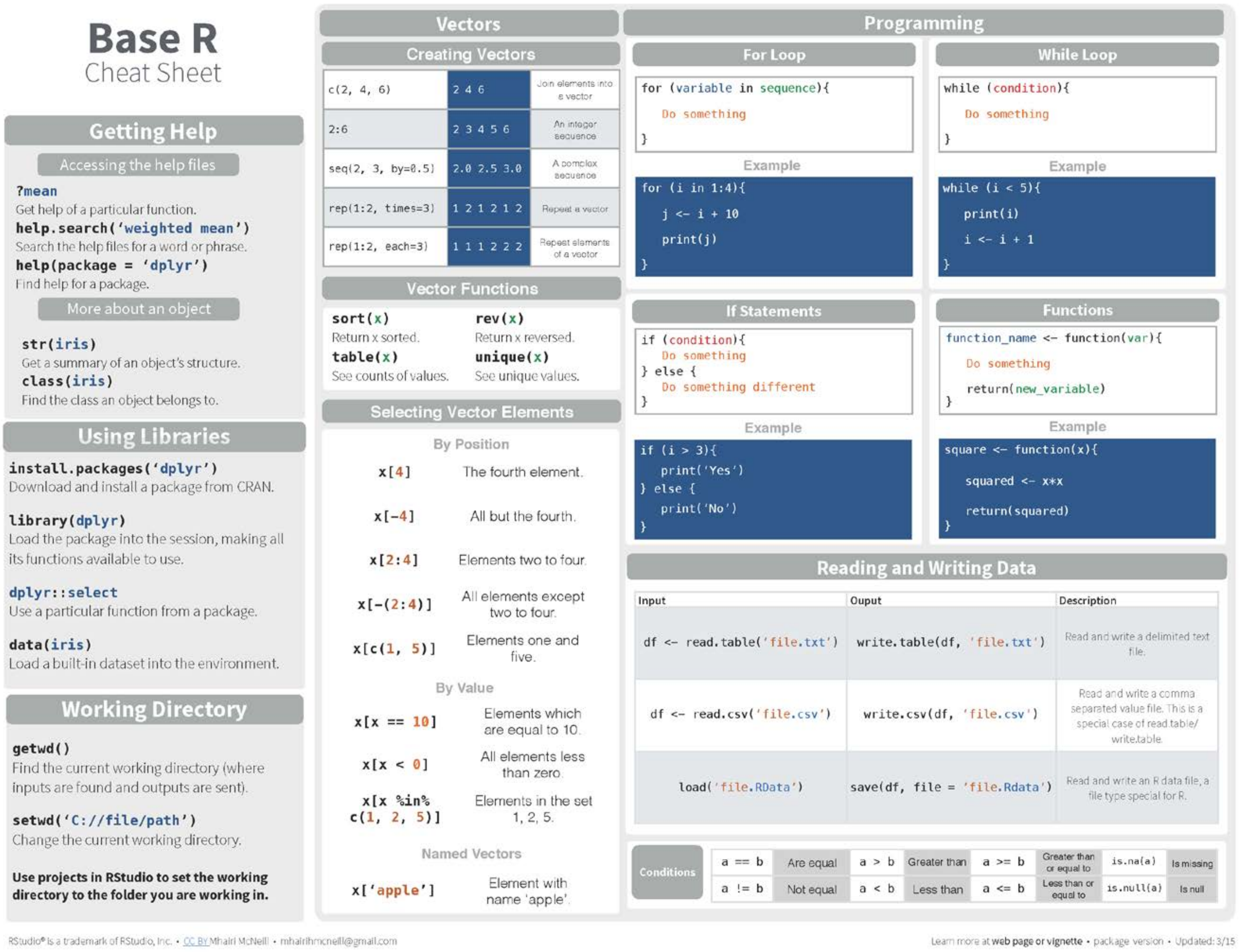


Figure 1: CRAN Mirrors and Packages



Base R
Cheat Sheet

Getting Help

Accessing the help files

?mean
Get help of a particular function.

help.search('weighted mean')
Search the help files for a word or phrase.

help(package = 'dplyr')
Find help for a package.

More about an object

str(iris)
Get a summary of an object's structure.

class(iris)
Find the class an object belongs to.

Using Libraries

install.packages('dplyr')
Download and install a package from CRAN.

library(dplyr)
Load the package into the session, making all its functions available to use.

dplyr::select
Use a particular function from a package.

data(iris)
Load a built-in dataset into the environment.

Working Directory

getwd()
Find the current working directory (where inputs are found and outputs are sent).

setwd('C://file/path')
Change the current working directory.

Use projects in RStudio to set the working directory to the folder you are working in.

Vectors

Creating Vectors

`c(2, 4, 6)`	2 4 6	Join elements into a vector
`2:6`	2 3 4 5 6	An integer sequence
`seq(2, 3, by=0.5)`	2.0 2.5 3.0	A complex sequence
`rep(1:2, times=3)`	1 2 1 2 1 2	Repeat a vector
`rep(1:2, each=3)`	1 1 1 2 2 2	Repeat elements of a vector

Vector Functions

sort(x) Return x sorted.	**rev(x)** Return x reversed.
table(x) See counts of values.	**unique(x)** See unique values.

Selecting Vector Elements

By Position

`x[4]`	The fourth element.
`x[-4]`	All but the fourth.
`x[2:4]`	Elements two to four.
`x[-(2:4)]`	All elements except two to four.
`x[c(1, 5)]`	Elements one and five.

By Value

`x[x == 10]`	Elements which are equal to 10.
`x[x < 0]`	All elements less than zero.
`x[x %in% c(1, 2, 5)]`	Elements in the set 1, 2, 5.

Named Vectors

`x['apple']`	Element with name 'apple'.

Programming

For Loop

```
for (variable in sequence){
  Do something
}
```

Example

```
for (i in 1:4){
  j <- i + 10
  print(j)
}
```

While Loop

```
while (condition){
  Do something
}
```

Example

```
while (i < 5){
  print(i)
  i <- i + 1
}
```

If Statements

```
if (condition){
  Do something
} else {
  Do something different
}
```

Example

```
if (i > 3){
  print('Yes')
} else {
  print('No')
}
```

Functions

```
function_name <- function(var){
  Do something
  return(new_variable)
}
```

Example

```
square <- function(x){
  squared <- x*x
  return(squared)
}
```

Reading and Writing Data

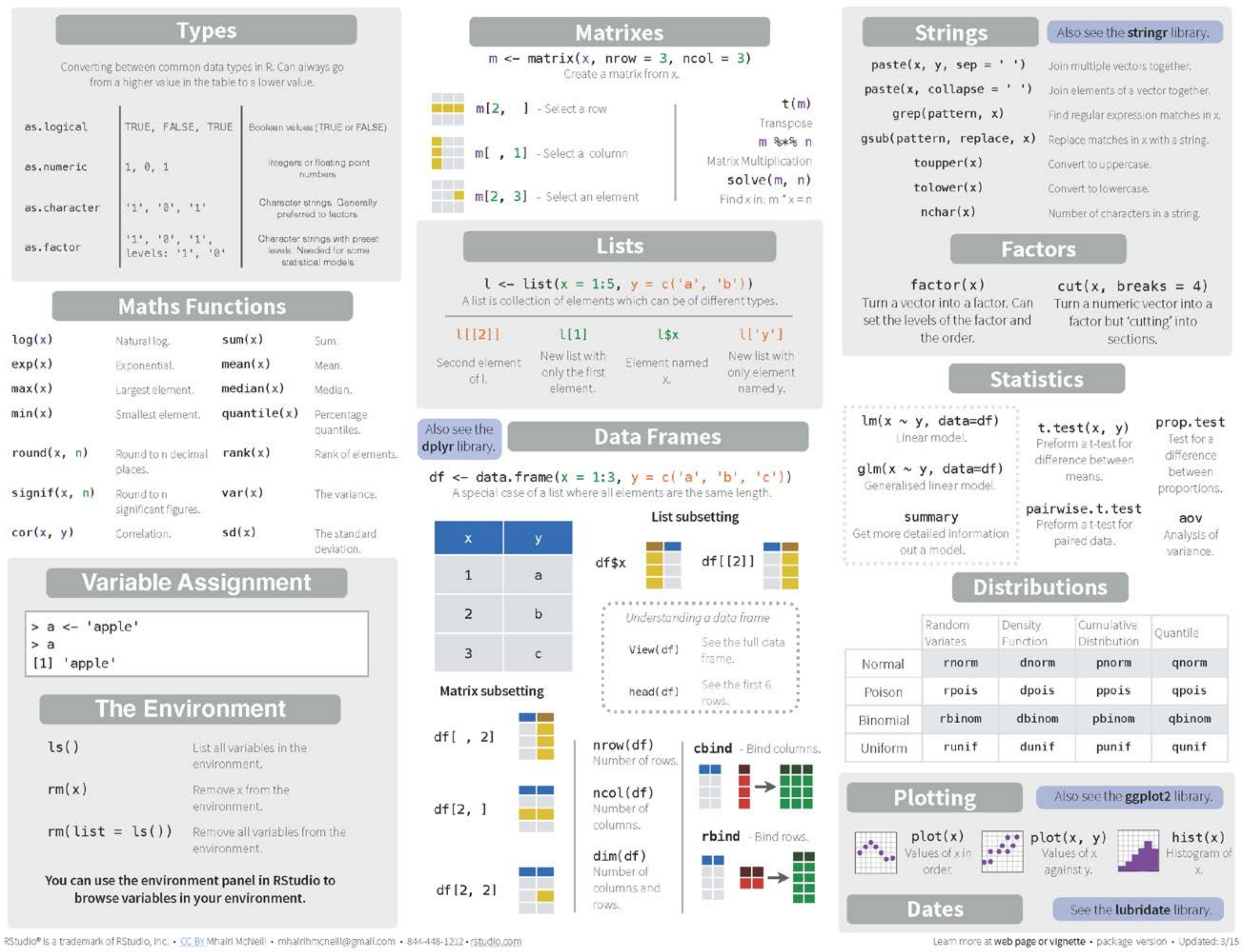
Input	Ouput	Description
`df <- read.table('file.txt')`	`write.table(df, 'file.txt')`	Read and write a delimited text file.
`df <- read.csv('file.csv')`	`write.csv(df, 'file.csv')`	Read and write a comma separated value file. This is a special case of read.table/ write.table.
`load('file.RData')`	`save(df, file = 'file.Rdata')`	Read and write an R data file, a file type special for R.

Conditions

`a == b`	Are equal	`a > b`	Greater than	`a >= b`	Greater than or equal to	`is.na(a)`	Is missing
`a != b`	Not equal	`a < b`	Less than	`a <= b`	Less than or equal to	`is.null(a)`	Is null

RStudio® is a trademark of RStudio, Inc. • CC BY Mhairi McNeill • mhairihmcneill@gmail.com

Learn more at **web page or vignette** • package version • Updated: 3/15



Types

Converting between common data types in R. Can always go from a higher value in the table to a lower value.

`as.logical`	`TRUE, FALSE, TRUE`	Boolean values (TRUE or FALSE)
`as.numeric`	`1, 0, 1`	Integers or floating point numbers
`as.character`	`'1', '0', '1'`	Character strings. Generally preferred to factors
`as.factor`	`'1', '0', '1', levels: '1', '0'`	Character strings with preset levels. Needed for some statistical models

Maths Functions

`log(x)`	Natural log.	`sum(x)`	Sum.
`exp(x)`	Exponential.	`mean(x)`	Mean.
`max(x)`	Largest element.	`median(x)`	Median.
`min(x)`	Smallest element.	`quantile(x)`	Percentage quantiles.
`round(x, n)`	Round to n decimal places.	`rank(x)`	Rank of elements.
`signif(x, n)`	Round to n significant figures.	`var(x)`	The variance.
`cor(x, y)`	Correlation.	`sd(x)`	The standard deviation.

Variable Assignment

```
> a <- 'apple'
> a
[1] 'apple'
```

The Environment

`ls()`	List all variables in the environment.
`rm(x)`	Remove x from the environment.
`rm(list = ls())`	Remove all variables from the environment.

You can use the environment panel in RStudio to browse variables in your environment.

Matrixes

`m <- matrix(x, nrow = 3, ncol = 3)`
Create a matrix from x.

`m[2, ]` - Select a row	`t(m)` Transpose
`m[ , 1]` - Select a column	`m %*% n` Matrix Multiplication
`m[2, 3]` - Select an element	`solve(m, n)` Find x in: m * x = n

Lists

`l <- list(x = 1:5, y = c('a', 'b'))`
A list is collection of elements which can be of different types.

`l[[2]]`	`l[1]`	`l$x`	`l['y']`
Second element of l.	New list with only the first element.	Element named x.	New list with only element named y.

Also see the **dplyr** library.

Data Frames

`df <- data.frame(x = 1:3, y = c('a', 'b', 'c'))`
A special case of a list where all elements are the same length.

x	y
1	a
2	b
3	c

List subsetting

`df$x` `df[[2]]`

Understanding a data frame

`View(df)`	See the full data frame.
`head(df)`	See the first 6 rows.

Matrix subsetting

`df[ , 2]`

`df[2, ]`

`df[2, 2]`

`nrow(df)`	Number of rows.
`ncol(df)`	Number of columns.
`dim(df)`	Number of columns and rows.

cbind - Bind columns.

rbind - Bind rows.

Strings

Also see the **stringr** library.

`paste(x, y, sep = ' ')`	Join multiple vectors together.
`paste(x, collapse = ' ')`	Join elements of a vector together.
`grep(pattern, x)`	Find regular expression matches in x.
`gsub(pattern, replace, x)`	Replace matches in x with a string.
`toupper(x)`	Convert to uppercase.
`tolower(x)`	Convert to lowercase.
`nchar(x)`	Number of characters in a string.

Factors

`factor(x)`
Turn a vector into a factor. Can set the levels of the factor and the order.

`cut(x, breaks = 4)`
Turn a numeric vector into a factor but 'cutting' into sections.

Statistics

`lm(x ~ y, data=df)`
Linear model.

`glm(x ~ y, data=df)`
Generalised linear model.

`summary`
Get more detailed information out a model.

`t.test(x, y)`
Preform a t-test for difference between means.

`pairwise.t.test`
Preform a t-test for paired data.

`prop.test`
Test for a difference between proportions.

`aov`
Analysis of variance.

Distributions

	Random Variates	Density Function	Cumulative Distribution	Quantile
Normal	`rnorm`	`dnorm`	`pnorm`	`qnorm`
Poison	`rpois`	`dpois`	`ppois`	`qpois`
Binomial	`rbinom`	`dbinom`	`pbinom`	`qbinom`
Uniform	`runif`	`dunif`	`punif`	`qunif`

Plotting

Also see the **ggplot2** library.

`plot(x)`
Values of x in order.

`plot(x, y)`
Values of x against y.

`hist(x)`
Histogram of x.

Dates

See the **lubridate** library.

Working with Manually Entered Excel Data

"In our lust for measurement, we frequently measure that which we can rather than that which we wish to measure... and forget that there is a difference." George Udny Yule

Typical Project Layout

Below is the directory layout of the project we will be using. All of the Excel data which we will be examining is found in under the Data directory. We will be looking at two groups: **Injury Control (IC)** and **Treated (T)**. Each study group has its own subdirectory, and all the files in the group directory belong to that group. In this dataset, the first injury control subject in this dataset is in file "IC1 Vitals.xlsx". The animal ID is in this Excel file, together with lab results, demographic information and experiment results.

```
Analysis\05JUN2017\
Animal transfer requests\
Charts, forms, supply list\
Data\Animal tally.xls
Data\Injury Control\IC1 Vitals.xlsx
Data\Injury Control\IC2 Vitals.xlsx
Data\Injury Control\IC3 Vitals.xlsx
Data\Injury Control\IC4 Vitals.xlsx
Data\Injury Control\IC5 Vitals.xlsx
Data\Injury Control\IC6 Vitals.xlsx
Data\Injury Control\IC7 Vitals.xlsx
Data\Injury Control\IC8 Vitals.xlsx
Data\Injury Control\IC9 Vitals.xlsx
Data\Treated\T1 Vitals.xlsx
Data\Treated\T2 Vitals.xlsx
Data\Treated\T3 Vitals.xlsx
Data\Treated\T4 Vitals.xlsx
Data\Treated\T5 Vitals.xlsx
Data\Treated\T6 Vitals.xlsx
Data\Treated\T7 Vitals.xlsx
Data\Treated\T8 Vitals.xlsx
Data\Treated\T9 Vitals.xlsx
Protocol and Addenda\
Meetings and Publications\
RECAP TEE\
RECAP carotid-flow\
RECAP idea\
Reports\
Surgical Reports\
```

The Project Setup File

To capture the project layout and location of data files, we create a file called “setup.r” that contains variables to describe the location of resources to be used for this analysis. This file will serve a central role by importing external libraries, declaring directories, providing common utility routines used by this project. It may also provide additional meta-information such as study groups or subjects to be included or excluded. We begin the "setup.r” file as follows:

```
#
# Minimum R version 2.4.0
#

# readxl is a light-weight read-only library for excel workbooks found in later version of R.
# Our usage of this library requires R 2.4.0 or later.

library( readxl )

#
# Output files should be dated...don't overwrite older analysis data
#
today <- Sys.Date()

server_share <- "\\\\ameda7aisr0107\\ISR_CANCIOLAB_4"
xlsx_dir <- paste( server_share, "\\00-PROTOCOLS\\A-16-004 RECAP\\Data\\", sep="")
idea_dir <- paste( server_share, "\\00-PROTOCOLS\\A-16-004 RECAP\\RECAP idea\\", sep="")
carotid_dir <- paste( server_share, "\\00-PROTOCOLS\\A-16-004 RECAP\\RECAP carotid-flow\\", sep="")

# include today’s date in the analysis output directory name
analysis_dir <- paste( server_share, "\\00-PROTOCOLS\\A-16-004 RECAP\\Analysis\\", today, sep="")
```

Common Utility Routines

Common utility routines should be placed in "setup.r". Notice that all the Excel vitals files are in the directory identified by the variable xlsx_dir. Our next step is therefore to read this directory and import each Excel file into the indicated study group. We will find a couple of utility functions to be useful. Our first utility routine will be to read individual columns:

```
#'
#'  Read a single column from an Excel spreadsheet.
#'
#' @param f            The Excel file to read
#' @param sheet_name   The Excel sheet name to read
#' @param col_idx      The index(es) of the column(s) to read
#'
#' @return A dataset consisting of a single column, formatted as a string.
#'
#' @export

read_column <- function( f, sheet_name, col_idx ) {

    col <- read_xlsx( f,     # if using the xlsx library, these are the arguments: f
        sheet=sheet_name,     # sheetName=sheet_name,
        col_names=FALSE,      # header=FALSE,
        skip=4,               # startRow=5,
                              # colIndex=col_idx,
                              # stringsAsFactors=FALSE,
        col_types="text"      # colClasses=c( "character")
        )
    #
    # reduce from a dataframe to an array
    #
    col <- as.character(unlist(col[,col_idx]))            #      col <- col[,1]

    #
    # find the last value that is not NA, and truncate array to this length
    #
    n <- max(c(1:length(col))[!is.na(col)])
    length(col) <- n

    # return this value as an array

    col
}
```

The second utility routine will provide a method to read a single row. The utility of these two routines will be shown below.

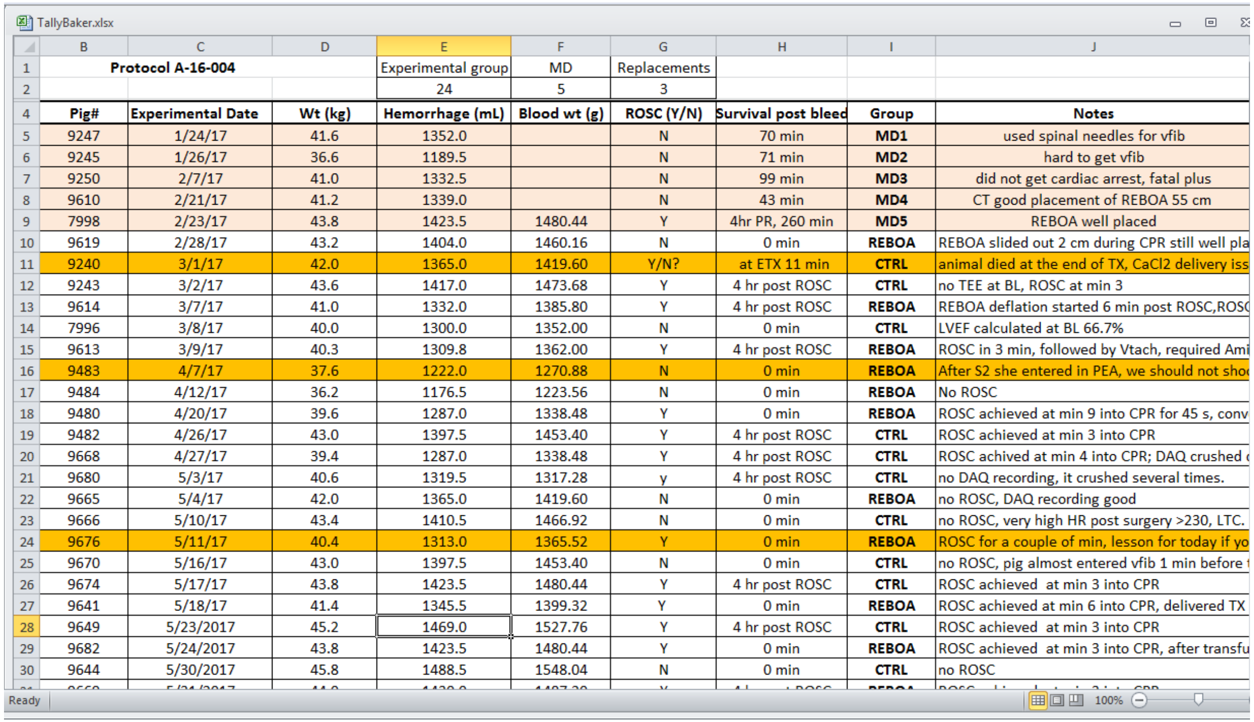
```
#' Read a single row from an Excel spreadsheet.
#'
#' Sometimes it is necessary to read a single row from an Excel spreadsheet.  A common use case is
#' when the column header information is not on the first row, or there are various rows between
#' the column header and the actual data.
#'
#' @param f            The Excel file to read
#' @param sheet_name   The Excel sheet name to read
#' @param row_idx      The index of the row to read.  Must be a single value.
#'
#' @return A character array consisting of a single row, without the first two columns.
#'
#' @export

read_row <- function( f, sheet_name, row_idx ) {

    row <- read_xlsx( f,     # if using the xlsx library, these are the arguments: f
        sheet=sheet_name,    # sheetName=sheet_name,
        col_names=FALSE,     # header=FALSE,
        skip=row_idx-1,      # startRow=row_idx,
        n_max=1,             # endRow=row_idx,
                                 # colIndex=c(3:99),
                                 # stringsAsFactors=FALSE,
        col_types="text"     # colClasses=c( "character")
        )

    # convert to array/vector
    as.character(row[,c(3:ncol(row))])               # Skip the first two columns
}
```

Code Documentation

In the previous section, you should have noticed the more verbose comment style and markers such as @param and @return. This style of comment is preceded by the two character combination #', and is used by R to generate code documentation for the online help system. This kind of comment is a best practice, but discussion and understanding of this comment style is beyond the purpose and scope of this document. For now, it is sufficient to simply understand this more formalized comment style is both understandable without additional information, as well as optional.

The Tally Sheet

A single master tally is usually used to keep track of which animals have been included in a study, to which study group the animal was added, and demographic type information. A sample tally sheet is shown below in Figure 2.



TallyBaker.xlsx

	B	C	D	E	F	G	H	I	J
1	Protocol A-16-004			Experimental group	MD	Replacements			
2				24	5	3			
4	**Pig#**	**Experimental Date**	**Wt (kg)**	**Hemorrhage (mL)**	**Blood wt (g)**	**ROSC (Y/N)**	**Survival post bleed**	**Group**	**Notes**
5	9247	1/24/17	41.6	1352.0		N	70 min	**MD1**	used spinal needles for vfib
6	9245	1/26/17	36.6	1189.5		N	71 min	**MD2**	hard to get vfib
7	9250	2/7/17	41.0	1332.5		N	99 min	**MD3**	did not get cardiac arrest, fatal plus
8	9610	2/21/17	41.2	1339.0		N	43 min	**MD4**	CT good placement of REBOA 55 cm
9	7998	2/23/17	43.8	1423.5	1480.44	Y	4hr PR, 260 min	**MD5**	REBOA well placed
10	9619	2/28/17	43.2	1404.0	1460.16	N	0 min	**REBOA**	REBOA slided out 2 cm during CPR still well pla
11	9240	3/1/17	42.0	1365.0	1419.60	Y/N?	at ETX 11 min	**CTRL**	animal died at the end of TX, CaCl2 delivery iss
12	9243	3/2/17	43.6	1417.0	1473.68	Y	4 hr post ROSC	**CTRL**	no TEE at BL, ROSC at min 3
13	9614	3/7/17	41.0	1332.0	1385.80	Y	4 hr post ROSC	**REBOA**	REBOA deflation started 6 min post ROSC,ROS
14	7996	3/8/17	40.0	1300.0	1352.00	N	0 min	**CTRL**	LVEF calculated at BL 66.7%
15	9613	3/9/17	40.3	1309.8	1362.00	Y	4 hr post ROSC	**REBOA**	ROSC in 3 min, followed by Vtach, required Ami
16	9483	4/7/17	37.6	1222.0	1270.88	N	0 min	**REBOA**	After S2 she entered in PEA, we should not sho
17	9484	4/12/17	36.2	1176.5	1223.56	N	0 min	**REBOA**	No ROSC
18	9480	4/20/17	39.6	1287.0	1338.48	Y	0 min	**REBOA**	ROSC achieved at min 9 into CPR for 45 s, conv
19	9482	4/26/17	43.0	1397.5	1453.40	Y	4 hr post ROSC	**CTRL**	ROSC achieved at min 3 into CPR
20	9668	4/27/17	39.4	1287.0	1338.48	Y	4 hr post ROSC	**CTRL**	ROSC achived at min 4 into CPR; DAQ crushed
21	9680	5/3/17	40.6	1319.5	1317.28	y	4 hr post ROSC	**CTRL**	no DAQ recording, it crushed several times.
22	9665	5/4/17	42.0	1365.0	1419.60	N	0 min	**REBOA**	no ROSC, DAQ recording good
23	9666	5/10/17	43.4	1410.5	1466.92	N	0 min	**CTRL**	no ROSC, very high HR post surgery >230, LTC.
24	9676	5/11/17	40.4	1313.0	1365.52	Y	0 min	**REBOA**	ROSC for a couple of min, lesson for today if yo
25	9670	5/16/17	43.0	1397.5	1453.40	N	0 min	**CTRL**	no ROSC, pig almost entered vfib 1 min before
26	9674	5/17/17	43.8	1423.5	1480.44	Y	4 hr post ROSC	**CTRL**	ROSC achieved at min 3 into CPR
27	9641	5/18/17	41.4	1345.5	1399.32	Y	0 min	**REBOA**	ROSC achieved at min 6 into CPR, delivered TX
28	9649	5/23/2017	45.2	1469.0	1527.76	Y	4 hr post ROSC	**CTRL**	ROSC achieved at min 3 into CPR
29	9682	5/24/2017	43.8	1423.5	1480.44	Y	0 min	**REBOA**	ROSC achieved at min 3 into CPR, after transfu
30	9644	5/30/2017	45.8	1488.5	1548.04	N	0 min	**CTRL**	no ROSC

Ready

Figure 2: Typical Tally Sheet

The tally sheet is simple enough to be read with a single call to read_xlsx. We convert the tally to a simpler data frame, then rename some columns for convenient access.

```
# this code depends on variables assigned in "setup.r", described above

#
# read tally file
#

tally_file <- paste( xlsx_dir, "..\\", "A-16-004_animal tally.xlsx", sep="" )
tally_file <- paste( xlsx_dir, "..\\", "TallyBaker.xlsx", sep="" )

tally <- read_xlsx( tally_file, "Animal Tally", skip=3 )
tally_df <- as.data.frame(tally,stringsAsFactors=FALSE)

#
# rename some columns
#

names( tally_df )[ names(tally_df) == "Pig#" ] <- "animal_id"
names( tally_df )[ names(tally_df) == "Experimental Date" ] <- "dt"
names( tally_df )[ names(tally_df) == "Wt (kg)" ] <- "weight"
names( tally_df )[ names(tally_df) == "Hemorrhage (mL)" ] <- "hemorrhage"
names( tally_df )[ names(tally_df) == "Blood wt (g)" ] <- "blood_wt"
names( tally_df )[ names(tally_df) == "ROSC (Y/N)" ] <- "ROSC"
```

For this analysis, we need the study date to adjust the clock times that were read from the individual Excel vitals charts. This is only needed because the data entry personnel don't have a standard place to put the study date in the individual sheets. They only record the time of each time point, not the date. To further complicate matters, Excel stores dates and times as a floating point numbers. Each day is 1 unit. For example, the value 0.25 would be 06:00 AM, and 0.45 would be close to 11:00 AM.

```
#'
#' Add a date to the clock times from the Excel Vitals file
#'
#' Vitals files contain the experiment time, but often do not contain the date.
#' This routine corrects the Vitals dataframe by adding the expermiment data
#' read from the tally file.
#'
#' @param tally_df     The tally dataframe read previously
#' @param mydf         A dataframe with a time columns: TM
#'
#' @return
#'
#' @export

fix_clock_times <- function( tally_df, mydf ) {
    mydf$TM[ mydf$TM == "DEATH" ] <- NA
    mydf$TM[ mydf$TM == "Death" ] <- NA

    HR <- as.numeric( mydf$TM ) * 24
    MM <- trunc(60*(HR - trunc(HR)))
    mydf$TOD <- sprintf( "%2.0f:%0.2d", trunc(HR),MM )
    mydf$TOD[ mydf$TOD == "NA:NA" ] <- ""

    animals <- unique( tally_df$animal_id )

    mydf$dt <- NA
    for( animal in animals ) {
        dt <- subset(tally_df, animal_id==animal)$dt
        mydf$dt[ mydf$animal_id == animal ] <- as.character(dt)
    }
    mydf$dt <- strptime( paste(mydf$dt, mydf$TOD), "%Y-%m-%d %H:%M" )

    mydf
}
```

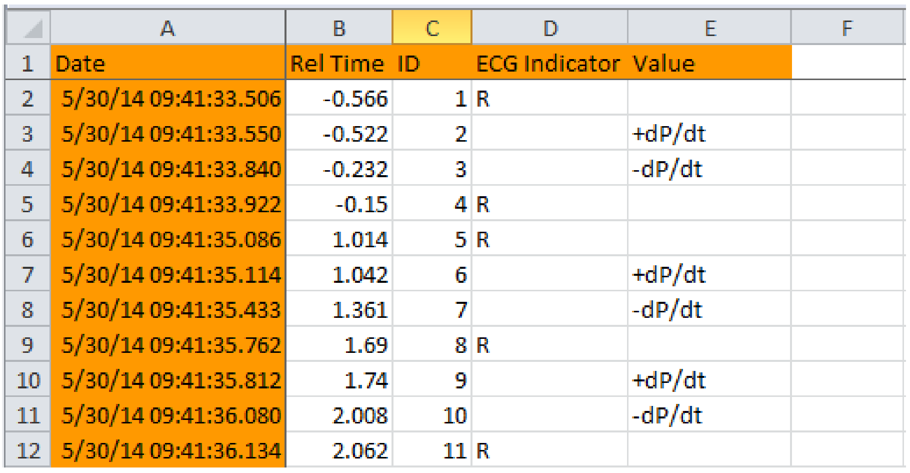
With this short function, it is now possible to convert the clock times stored in Excel floating point values to actual dates. Notice we make two special exceptions in the file: when the time is recorded as DEATH or Death, we don't use it. I've never been able to use DEATH as a time. Maybe it should be a time point instead? Perhaps? Hint.

```
vitals_df <- fix_clock_times( tally, vitals_df )
cbc_df <- fix_clock_times( tally, cbc_df )
```

Reading One Sheet of Standard Laboratory Data

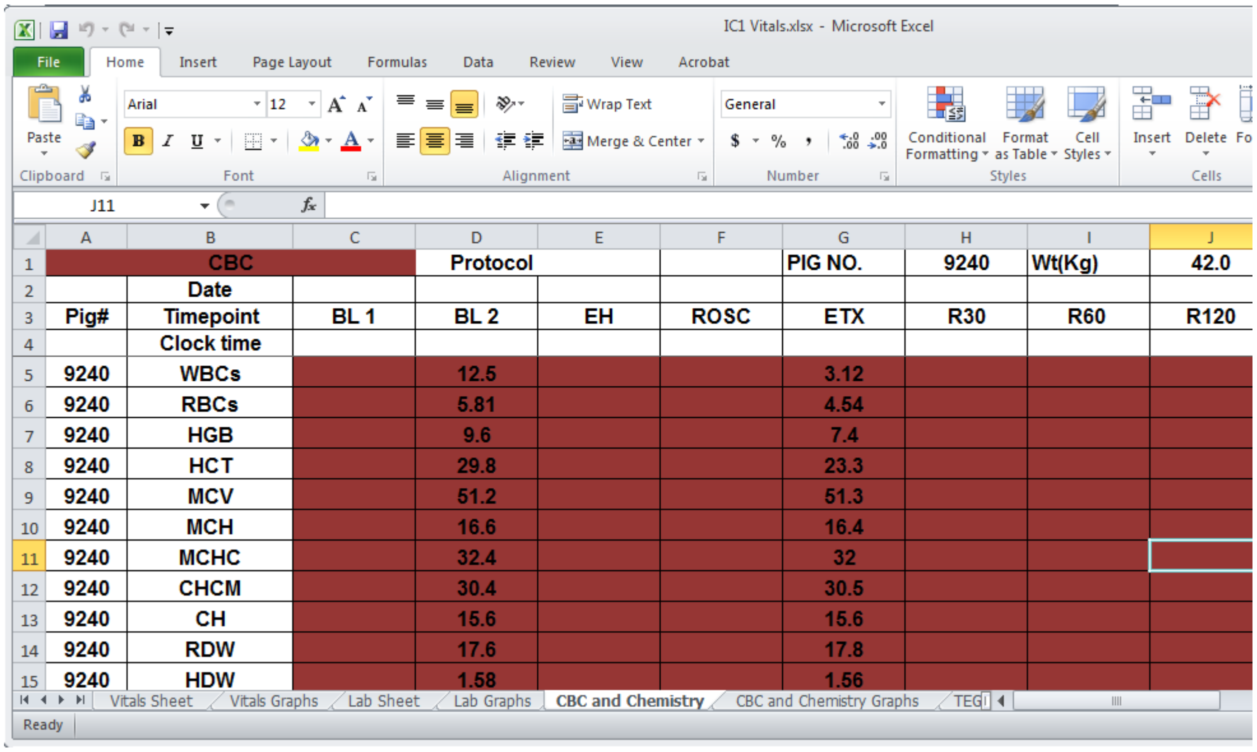
The Lab data we wish to read can be found in the "CBC and Chemistry" tab of our Excel workbook. By default, R expects the data to be in a particular format which most of us will recognize as a standard database format: column headers in the first row with sequential data following row-by-row. *(Figure 3: Desired Data Format)*

However, our CBC and Chemistry data is not in this format. The lab technicians typically enter the data with timepoints going across instead of down, chronological from left to right, as a traditional bed-chart. *(Figure 4)*



	A	B	C	D	E	F
1	Date	Rel Time	ID	ECG Indicator	Value	
2	5/30/14 09:41:33.506	-0.566	1	R		
3	5/30/14 09:41:33.550	-0.522	2		+dP/dt	
4	5/30/14 09:41:33.840	-0.232	3		-dP/dt	
5	5/30/14 09:41:33.922	-0.15	4	R		
6	5/30/14 09:41:35.086	1.014	5	R		
7	5/30/14 09:41:35.114	1.042	6		+dP/dt	
8	5/30/14 09:41:35.433	1.361	7		-dP/dt	
9	5/30/14 09:41:35.762	1.69	8	R		
10	5/30/14 09:41:35.812	1.74	9		+dP/dt	
11	5/30/14 09:41:36.080	2.008	10		-dP/dt	
12	5/30/14 09:41:36.134	2.062	11	R		

Figure 3: Desired Data Format



	A	B	C	D	E	F	G	H	I	J
1	CBC			Protocol			PIG NO.	9240	Wt(Kg)	42.0
2		Date								
3	Pig#	Timepoint	BL 1	BL 2	EH	ROSC	ETX	R30	R60	R120
4		Clock time								
5	9240	WBCs		12.5			3.12			
6	9240	RBCs		5.81			4.54			
7	9240	HGB		9.6			7.4			
8	9240	HCT		29.8			23.3			
9	9240	MCV		51.2			51.3			
10	9240	MCH		16.6			16.4			
11	9240	MCHC		32.4			32			
12	9240	CHCM		30.4			30.5			
13	9240	CH		15.6			15.6			
14	9240	RDW		17.6			17.8			
15	9240	HDW		1.58			1.56			

Figure 4: Actual Data Format

To read this data, we recognize the headings are in column B, the time points are in row 3, and the values of interest start at column C row 5. The following function, read_sheet_data, does just that:

it (1) reads the variable names in the second column, (2) reads the timepoint names in the third row, and (3) reads the data. Sometimes the number of rows/columns doesn't match the expected size due to appearance of empty rows or empty columns, so this routine has a couple of extra steps to ensure empty rows/columns do not otherwise interfere with the expected data.

```
#'
#' Read a vitals style sheet as a dataframe.
#'
#' The vitals sheet has event names on row three, and data beginning on row five.  The desired
#' column headers are in column B.  Therefore, the spreadsheet has the rows and columns
#' transposed from the structure we wish to see in a dataframe.
#'
#' @param f              The source Excel file.
#' @param sheet_name     The Excel sheet name to read.
#'
#' @return A dataframe containing the vitals (columns) for each study event (row).
#'
#' @export
read_sheet_as_dataframe <- function( f, sheet_name ) {
    cat("  reading cbc sheet: ", sheet_name, " subject file: ", subject, "\n" )

    #
    # variable names are in the second column
    #
    var_names <- read_column( f, sheet_name, 2 )

    #
    # timepoint names are in the third row
    #
    tp_names <- read_row( f, sheet_name, 3 )

    #
    # read main data portion as numberic
    #

    dat <- read_xlsx( f,     # if using the xlsx library, these are the arguments: read_xlsx2( f,
        sheet=sheet_name,    # sheetName=sheet_name,
        col_names=FALSE,     # header=FALSE,
        skip=4,              # startRow=5,
        n_max=length(var_names),   # endRow=4 + length(var_names),
                             #          colIndex=c(3:12),
                               # stringsAsFactors=FALSE,
        col_types=c( "text" ) # colClasses="character"
        )

    dat_df <- as.data.frame(dat)
    dat_df <- dat_df[,c(3:12)]

    length(tp_names) <- length(dat_df)

    #
    # swap rows and columns...flip...translate...
    #
    dat_t <- as.data.frame(t(dat_df))

    names(dat_t) <- var_names
    dat_t$TP <- tp_names

    dat_t
}
```

Repeating the Process: Reading an Entire Study

Since we have organized our groups by directory, we will use a short program to list all the files in each directory and extract data accordingly. The first sheet we will look at is "CBC and Chemistry". The end result will be an R ***dataframe*** that contains all CBC data for all animals from all sheets. As we read each sheet of data, we will attach a little bit of additional information to the dataframe indicating the source file, subject group, and animal ID. To finish the process, a few lines of code will clean up the automatically assigned row numbers and remove empty data columns.

```
#
# Read all Vitals sheets for all animals.
#

groups <- list.files( xlsx_dir )

cbc_df <- NULL
for( group in groups ) {
    cat("group: ", group, "\n" )

    group_dir <- paste( xlsx_dir, group, sep="" )
    group_files <- list.files( group_dir)

    for( subject in group_files ) {
        if( length(grep("mean", subject )) >= 1 ) {
            # ignore
            cat("ignoring file: ", subject)
        } else {
            f <- paste( group_dir, subject, sep="\\" )
            cbc <- read_sheet_as_dataframe( f, "CBC and Chemistry" )
            cbc$group <- group
            cbc$animal_id <- read_animal_id( f, "CBC and Chemistry" )
            cbc$file <- subject
            if( is.null( cbc_df ) ) {
                cbc_df <- cbc
            } else {
                tryCatch(
                    cbc_df <- rbind( cbc_df, cbc ),
                    error = function(e) {
                        cat( "ERROR: Column Names do not match\n", names(cbc),"\n", names(cbc_df), "\n" )
                        stop(e)
                    }
                )
            }
        }
    }
}
```

Post-Process and Data Cleanup

After importing a large dataset, several types of cleaning will be needed. The first type of clean will be consistency of data types and values. Using correct data types will make analysis and graphing much easier. For instance, the string "1.2.3" is not a valid number. In the post processes phase, mistakes like this will be found by converting data to the correct data types. Usually this involves converting strings to numbers and dates, but it also involves some handling of missing data.

Another post-processing step is ordering of "factors". A factor is a data type of enumerated values. For instance, the time points in a study will usually be "factors", often enumerated as "Baseline", "Injury", "Treatment", "R30", "R60", "R120", etc. And there will be a finite number of factors. Factors provide a convenient way to group points. The default order of factors is alphabetic, but any graph probably prefers chronologic order. The snippet below provides examples of removing empty columns, re-ordering factors, and converting a series of columns to numeric after replacing the text value "-" with NA and the text value "Off" with zero.

```
  #
  # post processing
  #

  # reset row numbers...simple cleaning
  rownames(full_df) <- NULL

  # remove columns where colname is NA
  full_df <- full_df[,!is.na(colnames(cbc_df))]

  # factors will be helpful here
  full_df$TP   <- factor(full_df$TP,   levels=c("BL 1","BL 2", "EH", "ROSC", "ETX", "R30", "R60", "R120",
"R180", "R240/D") )

  # convert data columns to numbers…indicating warnings if unexpected data
  options( warn=1 )
  for( i in c(1:(ncol(full_df) - 4)  )) {
      cat("process col: ", i, "\n" )

      u <- full_df[,i]
      u[u == "-"] <- NA
      u[u == "off"] <- 0
      class(u) <- "numeric"
      full_df[,i] <- u
      warnings()
  }
```

Some data errors can be easily found by applying consistency rules. For instance, MAP can never be greater than the systolic pressure, or less than the diastolic pressure. SpO2 can never be greater than 100.

Some simple tests to find erroneous data entry are given below:

```
# sanity checks

subset( vitals_df, MAP > SYS )
subset( vitals_df, MAP < DIA )
subset( vitals_df, CCO > 10 )
subset( vitals_df, SpO2 > 100 )
subset( vitals_df, SpO2 < 20 )
subset( vitals_df, RR > 40 )
subset( vitals_df, `PAP/S` > SYS )
subset( vitals_df, (TP == 'EH') & (MAP > 100) )
```

Each one of these queries found one or more errors in the data. We then went back to the Excel spreadsheets to examine the source of the error and fix it in the raw data source.

The Resulting Dataframe and Preliminary Analysis

The resulting R dataframe contains all CBC information for all animals in all spreadsheets. The ***head*** command provides a quick way to explicitly see the first few rows of data.

```
> head( cbc_df )
  WBCs RBCs  HGB  HCT  MCV                MCH MCHC CHCM   CH                RDW  HDW  PLT
1 <NA> <NA> <NA> <NA> <NA>               <NA> <NA> <NA> <NA>               <NA> <NA> <NA>
2 12.5 5.81  9.6 29.8 51.2 16.600000000000001 32.4 30.4 15.6 17.600000000000001 1.58  226
3 <NA> <NA> <NA> <NA> <NA>               <NA> <NA> <NA> <NA>               <NA> <NA> <NA>
4 <NA> <NA> <NA> <NA> <NA>               <NA> <NA> <NA> <NA>               <NA> <NA> <NA>
5 3.12 4.54  7.4 23.3 51.3 16.399999999999999   32 30.5 15.6               17.8 1.56  193
6 <NA> <NA> <NA> <NA> <NA>               <NA> <NA> <NA> <NA>               <NA> <NA> <NA>
                 MPV INITIALS Time Pt. Clock time  BUN  CKI CRE2                 TNI
1               <NA>     <NA>     BL 1       <NA> <NA> <NA> <NA>                <NA>
2                7.2     <NA>     BL 2       <NA>  6.9  432 0.97                0.06
3               <NA>     <NA>       EH       <NA> <NA> <NA> <NA>                <NA>
4               <NA>     <NA>     ROSC       <NA> <NA> <NA> <NA>                <NA>
5 8.3000000000000007     <NA>      ETX       <NA>  7.4  331 1.17 0.24099999999999999
6               <NA>     <NA>      R30       <NA> <NA> <NA> <NA>                <NA>
                  TBI  AST ALTI INITIALS.1   TP          group animal_id           file
1                <NA> <NA> <NA>       <NA> BL 1 Injury Control      9240 IC1 Vitals.xlsx
2 0.14000000000000001   20   64       <NA> BL 2 Injury Control      9240 IC1 Vitals.xlsx
3                <NA> <NA> <NA>       <NA>   EH Injury Control      9240 IC1 Vitals.xlsx
4                <NA> <NA> <NA>       <NA> ROSC Injury Control      9240 IC1 Vitals.xlsx
5                0.12   18   44       <NA>  ETX Injury Control      9240 IC1 Vitals.xlsx
6                <NA> <NA> <NA>       <NA>  R30 Injury Control      9240 IC1 Vitals.xlsx
>
```

For CBC's, all animals in this study should have data for the baseline (BL 2) and end of transfusion (ETX). A quick box plot of the RBC and WBC values should confirm our data is meaningful.

```
>
>
> boxplot( WBCs ~ TP, data=cbc_df, main="White Blood Cells"  )
```

Here we have elementary confirmation that our white blood cell count does indeed vary by time point. Although this is not really meaningful data at this point beyond our purpose of showing that we do indeed have viable data.

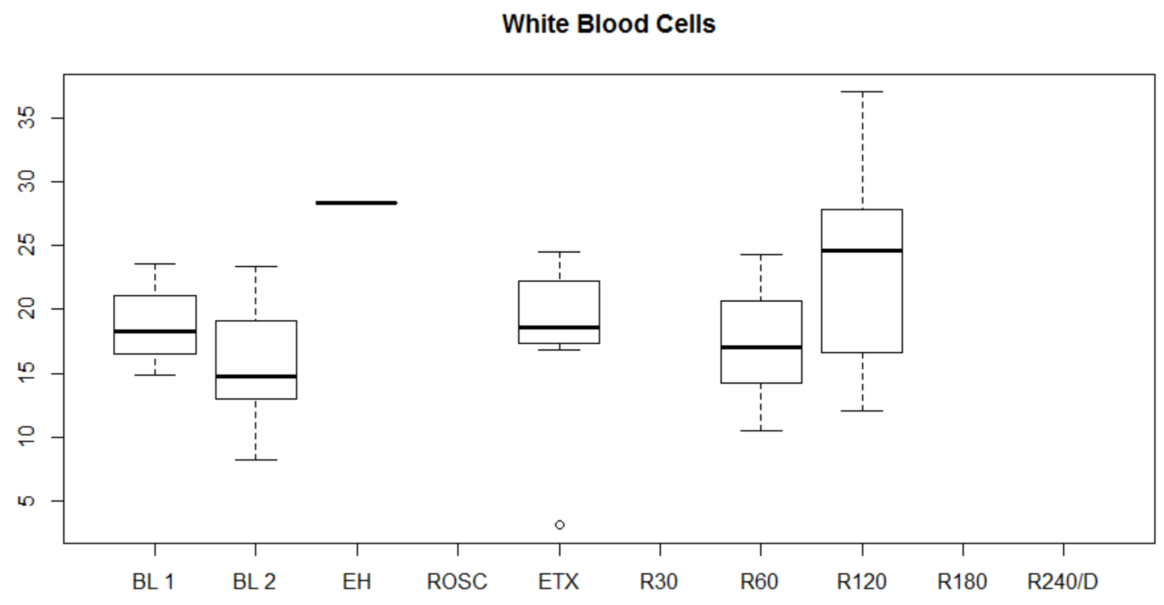


Figure 5: Boxplot of WBC by Time point from Excel Vitals Data

Advanced Topic: Prettier box plots

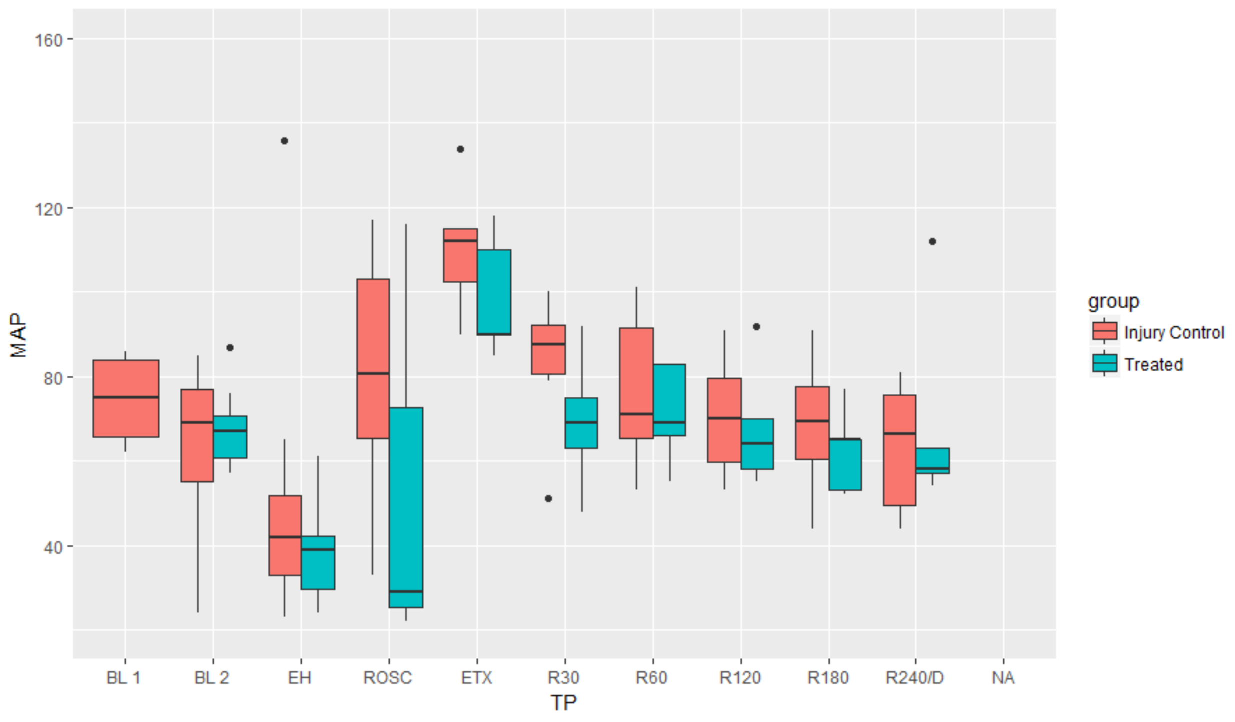


Figure 6: Enhanced Boxplot of WBC by Time point and Group

R is really good at making graphs. Simple graphs. Complex graphs. Domain specific graphs. Galleries of these images exist online. There are three commonly used graphics systems in R. Yes, that does make things complicated and, No, you don't need to learn all of them.

Start with the core graphs: plot(), boxplot(). If you need them, learn Lattice graphs: xyplot(), xy…

But if you want pretty graphs and are willing to work a little harder, GGPLOT2 is the ultimate solution. The above plot of MAP vs TP was created by the code below:

```
ggplot(vitals_df, aes(x=TP, y=MAP, fill=group)) +
    geom_boxplot() +
    coord_cartesian(ylim = c(20, 160))
```

There are entire books written on ggplot2. More information on ggplot2 can be found online, in books, and in the online help section. Information on this topic is extensive and beyond the scope of this document.

Advanced Topic: Code Layout and Storage

Just as project data files should be stored in a standard format, project analysis code should be stored as well in a standard format. Source code is typically stored in a revision control system. A revision control system keeps track not only of the current code, but also all changes made to the code since it was first created as well as who made those changes.

The most popular revision control system is GIT, and free accounts are available from internet suppliers such as github.com.

Automated Recording of Instrument Data

There are multiple instrument data collection systems in use at the ISR, and these systems are capable of recording data in multiple formats. One format of the ASCII data recordings was created by Guy Drew, and several version of it were used over time by his software. In particular, two of his file formats are commonly used for vitals and waveforms. The Guy Drew Vitals ASCII format consists of three rows of header data followed by a blank line, followed by a standard tab delineated dataset. Traditionally, these file names end in .VTL.

```
Dynamic Research Evaluation Workstation
Vitals Report
Thursday, August 27, 2009

Time    DSI-ABP-S   DSI-ABP-D    DSI-ABP-M       Temp-M  EKG-M   RateRate-EKG    Rate-DSI-ABP    MKR #
08:46:17        124.471 92.173  108.818 48.767  -0.236  71.571  71.571  71.698 0.000
08:46:22        123.575 91.359  107.833 48.766  -0.038  69.338  69.338  67.834 0.000
08:46:27        123.379 92.051  108.430 48.767  -0.129  64.849  64.849  66.294 0.000
```

The Guy Drew Waveform ASCII format consists of 16 rows of header information, followed by a blank line, followed by a standard tab delineated dataset.

```
Dynamic Research Evaluation Workstation
Analog Waveform Recording: v7.03mx
Data Format : Scaled ASCII (TXT)
Thursday, August 27, 2009 : 08:46:17

Company Name: U S Army
Organization: ISR
Study Title: Wade Darpa  A-07-006 TS5
Sugery Date: Thursday, August 27, 2009
Subject Number: 8521
Medical ID: 0
DAQ Operator(s):
Base File Name: DARPA 8521 082709
Sequence Number: 001
Sample Rate: 500
Notes: See Notes File.

EKG     DSI-ABP DSI-ECG Temp    MKR #
-0.049  94.739  0.034   48.769  0.000
-0.049  94.983  0.044   48.769  0.000
-0.073  94.739  0.034   48.769  0.000
-0.073  94.739  0.044   48.769  0.000
```

Reading Guy Drew File Headers

As seen from the above samples, these ASCII files use a verbose date format with unique parsing requirements. We can parse the date with the code below:

```
#' Parse the Guy Drew date
#'
#' @param line      A Date or Date+Time value used in Guy Drew waveform and vitals files.
#'
#' @return   a POSIXct date
#'
#' @export
parseGuyDrewDate <- function( line ) {
    ltime <- strptime( line, format="%A, %B %d, %Y %H:%M:%S" )
    if( is.na(ltime) ) {
        ltime <- strptime( line, format="%A, %B %d, %Y" )
    }
    ptime <- as.POSIXct( ltime )
    ptime
}
```

For either vitals or waveform datasets, the header can be read and parsed with the routine below:

```
#' Read DAQ Header
#'
#' This functions reads either a VTL or a DAT (waveform) header from Guy Drew ASCII Files.
#'
#' @param file      The ASCII VTL file to open
#'
#' @return          A list of the header variables including the end of the Guy Drew header.
#'                 The additional variable dataStart indicates where data reading should begin.
#'
#' @export
readHeader <- function( file ) {

    l <- list()
    lines <- readLines(file, n=20 )

    date_line = 3
    i <- 1
    h <- FALSE

    for( line in lines ) {
        var <- NULL
        if( h ) {
            l[["header"]] = line
            l[["ncols"]] = length( strsplit( line, "\t" )[[1]] )
            break
        }
        if( nchar(line) == 0 ) {
            if( i ==5 ) next                        # waveform files have a blank line under the date
            h <- TRUE                    # probably a vitals file
            next
        }

        if( i == 1 ) {
            if( line != "Dynamic Research Evaluation Workstation" ) {
                stop("Invalid Guy Drew ASCII file.")
            }
        } else if( i == 2 ) {
            if( startsWith( line, "Vitals Report") ) {
                l[["type"]] <- "vitals"
            } else if( startsWith( line, "Analog Waveform") ) {
                l[["type"]] <- "waveform"
            } else if( startsWith( line, "Notes") ) {
                l[["type"]] <- "notes"
            } else if( startsWith( line, "Electronic Lab Book") ) {
                l[["type"]] <- "book"
            } else {
                stop("Unknown header in Guy Drew ASCII file.")
            }
        } else if ( i == date_line ) {
            val <- parseGuyDrewDate( line )
            var <- "studyDate"
        } else {
            arr <- strsplit( line, ":" )[[1]]
            var <- gsub("[[:space:]]", "", arr[1] )
            val <- gsub("^[[:space:]]*", "", arr[2] )

            if( var == "SugeryDate") {
                var = "SurgeryDate"  # for programmers who cant spell
            }
            if( var == "DAQOperator(s)" ) {
                var = "DAQOperator"
            }
        }
        if( !is.null(var) ) {
            l[[var]] <- val
        }

        i <- i + 1
    }
    l[["dataStart"]] <- i

    l
}
```

Reading ASCII Vitals Files

Reading a single vitals data file is now directly done in two steps: read the header using readHeader, then use the header information to read the data portion.

```
> dir <- "Y:\\A-07-006 TS6 DARPA\\x - Archive\\DARPA Wade TS5\\DARPA TS5 8521"
> vtl <- "DARPA 8521 082709_090827_0846_v_001.vtl"
> h <- readHeader( paste(dir, vtl, sep="\\") )

> vitals <- read.csv(file, header=TRUE, skip=h$dataStart, sep="\t",
    colClasses=c("character", rep("numeric", h$ncols - 1) )
    )
```

The data is not perfect for two reasons. First, the Time column of vitals contains only the time, not the date. The study date is found in the header file. To get a well-defined time field, we need to combine these two values and parse them. The following two lines fix the date as needed.

```
> vitals$TM <- paste( format( h$studyDate, "%Y-%m-%d" ), vitals$Time )
> vitals$TM <- strptime( vitals$TM, "%Y-%m-%d %H:%M:%S" )
```

The second issue with the vitals files is that they are segmented into 5 minute segments, so we need to concatenate the results of all of the vitals files to obtain the desired study dataset. The resulting script is as follows:

```
dir <- "Y:\\A-07-006 TS6 DARPA\\x - Archive\\DARPA Wade TS5\\DARPA TS5 8521"
vtl <- "DARPA 8521 082709_090827_0846_v_001.vtl"

read_one_animal_vitals <- function ( animal_id ) {
    p <- paste( ".* ", animal_id, " .*.vtl", sep="" )
    files <- sort( list.files(path=dir, pattern=p, full.names=TRUE ) )

    vitals <- NULL
    for( file in files ) {
        h <- readHeader( paste(dir, vtl, sep="\\") )
        segment <- read.csv(file, header=TRUE, skip=h$dataStart, sep="\t",
            colClasses=c("character", rep("numeric", h$ncols - 1) )
            )
        segment$TM <- paste( format( h$studyDate, "%Y-%m-%d" ), segment$Time )
        segment$TM <- strptime( segment$TM, "%Y-%m-%d %H:%M:%S" )

        if( is.null(vitals) ) {
            vitals <- segment
        } else {
            vitals <- rbind( vitals, segment )
        }

    }

    vitals
}
```

Now we can enjoy the immediate gratification of a few plots. See Figure 7.

```
> d <- read_one_animal_vitals( "8521" )
> plot( d$TM, d$Rate.EKG, col='green', typ='l', xlab="Time", ylab="HR and ABP" )
> lines( d$TM, d$DSI.ABP.M, col='red' )
```

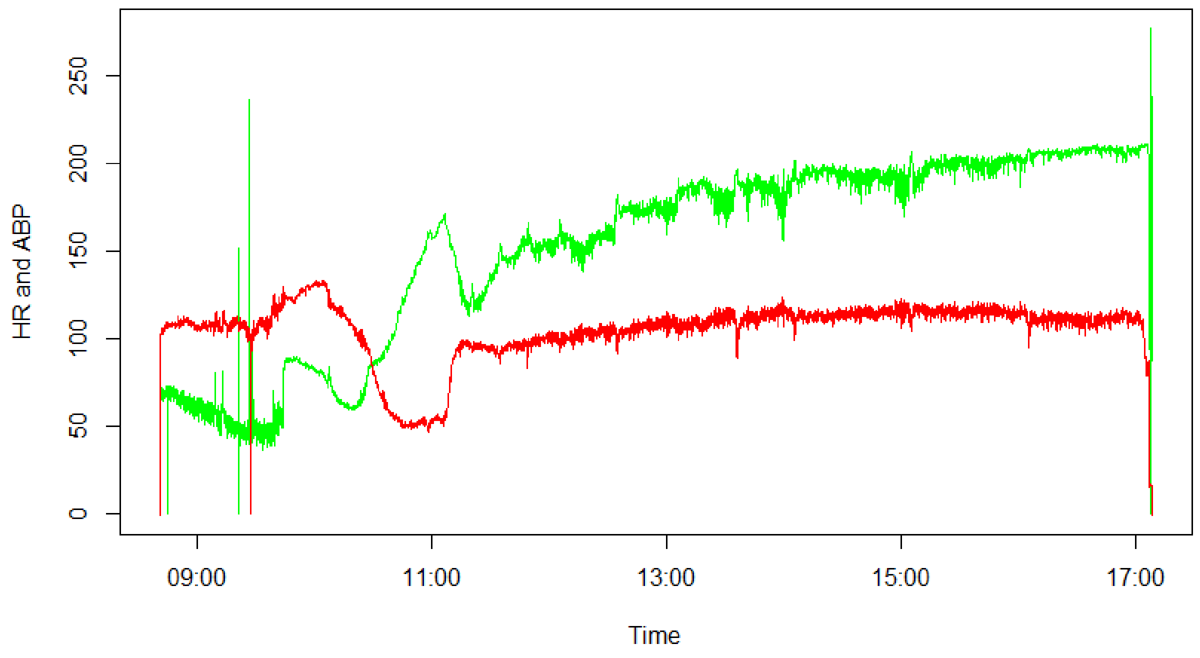


Figure 7: Plot of Continuous Vitals

Reading ASCII Notes File

The "notes" file should be in the same directory as the other data files. It can be found with the list.files command and read just as easily.

```
file <- list.files(path=dir, pattern=".*notes.*", full.names=TRUE )
h <- readHeader(file)
notes <- read.csv(file, header=TRUE, skip=h$dataStart, sep="\t",
    colClasses=c("character", "character", "character")
    )

# fix the time column of the notes as above...

notes$TM <- paste( format( h$studyDate, "%Y-%m-%d" ), notes$Time )
notes$TM <- strptime( notes$TM, "%Y-%m-%d %H:%M:%S" )
```

Now we can plot the results on top of the previous graph. For this plot, we will use ggplot. One reason for this is that the vertical line function in the base plots (abline) does not understand date/time values. The second reason is that ggplot is a superior plotting system, and this problem gives us a reason to explore it a little more.

```
tm0 <- d$TM[1]
d$mins <- as.numeric( difftime( d$TM, tm0, units="mins" ) )
notes$mins <- as.numeric( difftime( notes$TM, tm0, units="mins" ) )

ggplot(d, aes(x=mins,y=Rate.EKG)) +
        coord_cartesian( xlim=c(100,170), ylim=c(0,400) ) +
        xlab("") +
        ylab("HR and ABP" ) +
        ggtitle( "Sample Plot" ) +
        geom_line(color='green') +
        geom_line(data=d, y=d$DSI.ABP.M, color='red' ) +
        geom_vline(data=notes, xintercept=notes$mins, color="blue") +
        geom_text(data=notes, aes(x=notes$mins, y=300, label=notes$Notes), show.legend=FALSE,
color="black", angle=90)
```

In this plot, we have demonstrated many features of the ggplot library. Here are a few:

- Setting a plot title using ggtitle()
- Setting the horizontal and vertical graph range using xlim() and ylim()
- Labeling the axes using xlab() and ylab()
- Drawing multiple lines from different datasets using geom_line()
- Drawing vertical lines using geom_vline()
- Drawing rotated text on top of a graph using geom_text()

See Figure 8: Overlay of Time Points on Continuous Vitals.

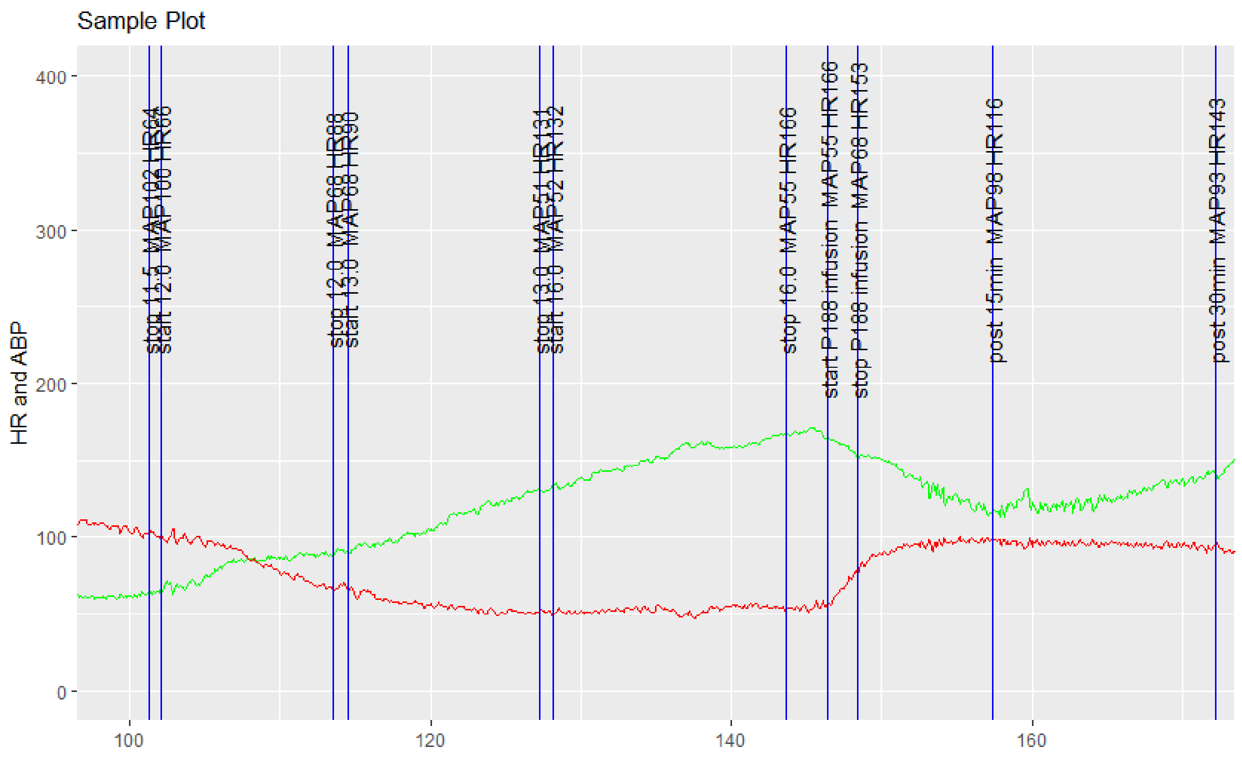


Figure 8: Overlay of Time Points on Continuous Vitals

Reading ASCII Wave Files

Our final task is to plot a simple waveform. Using the tools that we have already assembled, this is straightforward, and the plot is shown in Figure 9: EKG Waveform Signal.

```
dir <- "Y:\\A-07-006 TS6 DARPA\\x - Archive\\DARPA Wade TS5\\DARPA TS5 8521"
dat <- "DARPA 8521 082709_090827_1021_w_020.dat"
file <- paste(dir, dat, sep="\\")

h <- readHeader( file )
wf <- read.csv(file, header=TRUE, skip=h$dataStart, sep="\t",
    colClasses=c("numeric")
    )

plot( wf$EKG, typ='l', xlim=c(0,2000), xlab="Time (ms)", ylab='EKG', main="EKG / 8521" )
```

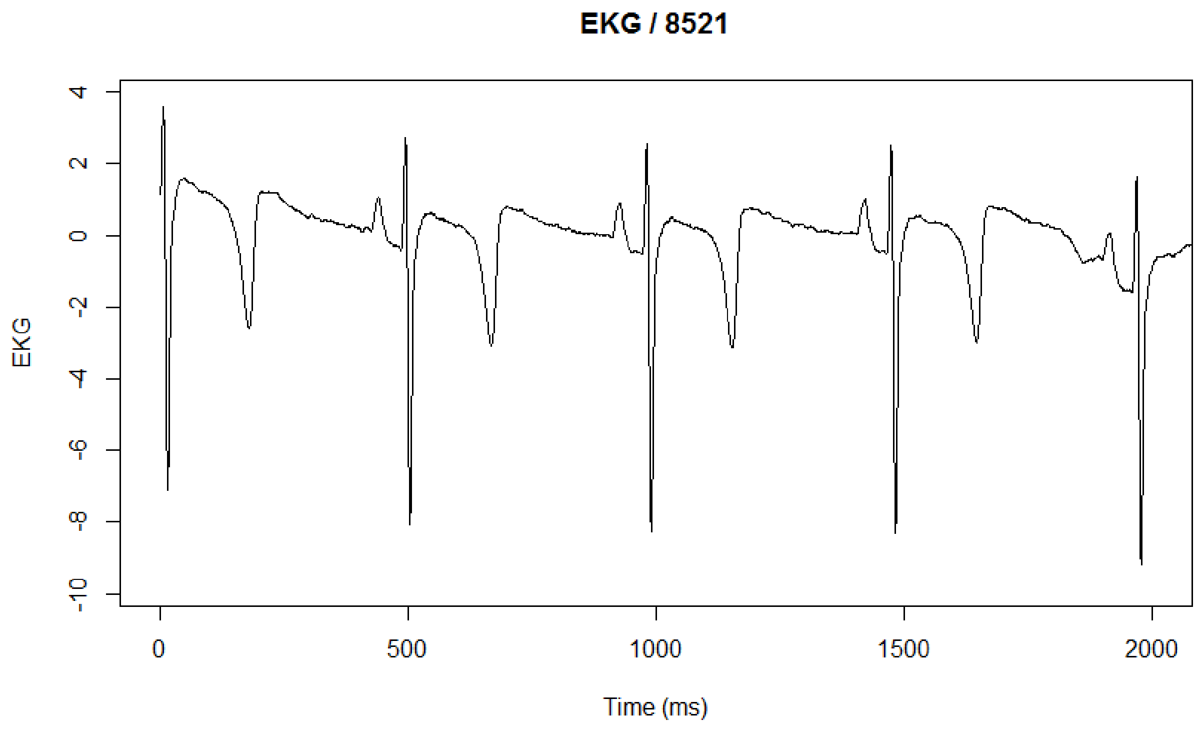
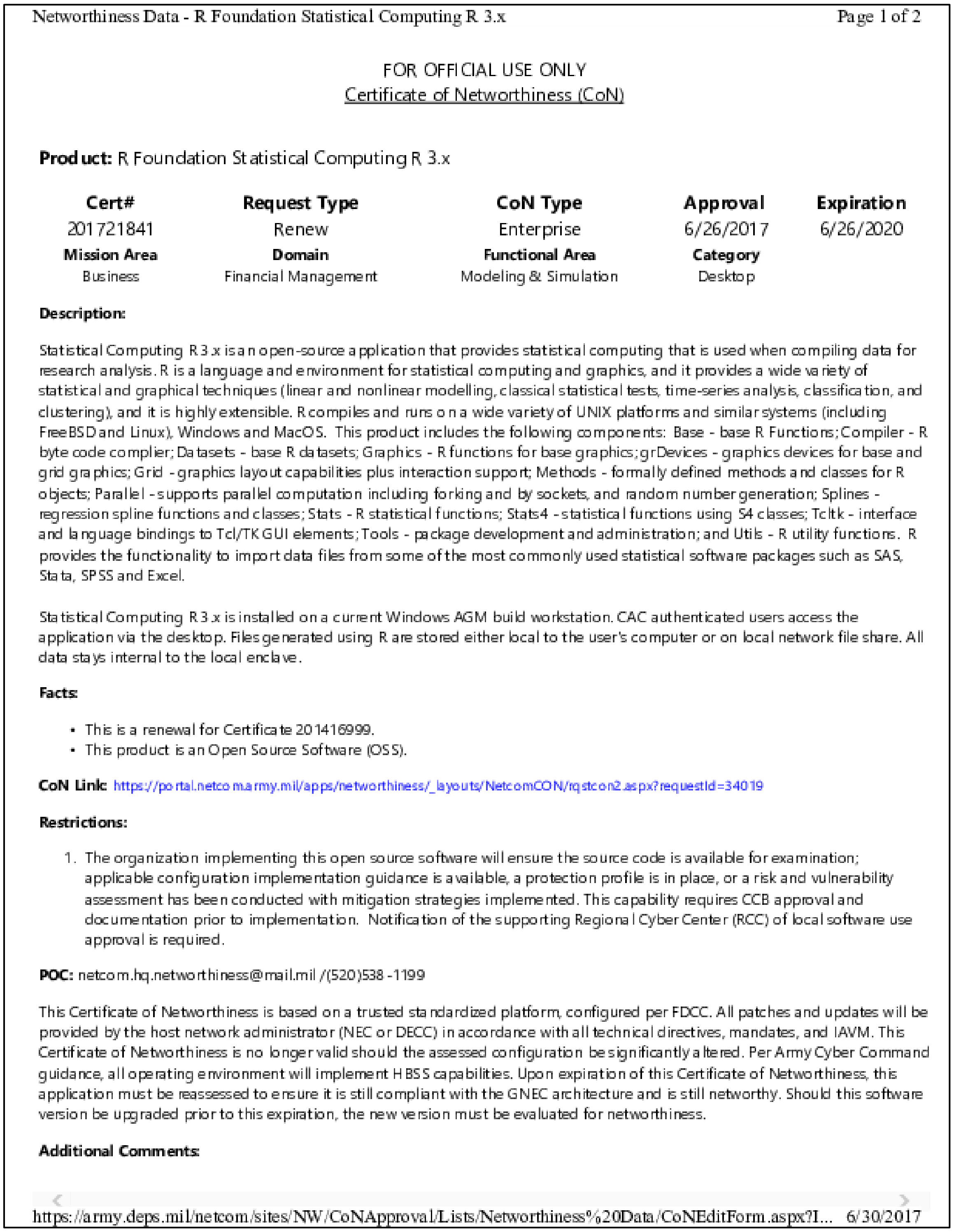


Figure 9: EKG Waveform Signal

DoD Security and Certificates of Networthiness



Networthiness Data - R Foundation Statistical Computing R 3.x Page 1 of 2

FOR OFFICIAL USE ONLY

Certificate of Networthiness (CoN)

Product: R Foundation Statistical Computing R 3.x

Cert#	Request Type	CoN Type	Approval	Expiration
201721841	Renew	Enterprise	6/26/2017	6/26/2020
Mission Area	**Domain**	**Functional Area**	**Category**	
Business	Financial Management	Modeling & Simulation	Desktop	

Description:

Statistical Computing R 3.x is an open-source application that provides statistical computing that is used when compiling data for research analysis. R is a language and environment for statistical computing and graphics, and it provides a wide variety of statistical and graphical techniques (linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, and clustering), and it is highly extensible. R compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS. This product includes the following components: Base - base R Functions; Compiler - R byte code complier; Datasets - base R datasets; Graphics - R functions for base graphics; grDevices - graphics devices for base and grid graphics; Grid - graphics layout capabilities plus interaction support; Methods - formally defined methods and classes for R objects; Parallel - supports parallel computation including forking and by sockets, and random number generation; Splines - regression spline functions and classes; Stats - R statistical functions; Stats4 - statistical functions using S4 classes; Tcltk - interface and language bindings to Tcl/TK GUI elements; Tools - package development and administration; and Utils - R utility functions. R provides the functionality to import data files from some of the most commonly used statistical software packages such as SAS, Stata, SPSS and Excel.

Statistical Computing R 3.x is installed on a current Windows AGM build workstation. CAC authenticated users access the application via the desktop. Files generated using R are stored either local to the user's computer or on local network file share. All data stays internal to the local enclave.

Facts:

- This is a renewal for Certificate 201416999.
- This product is an Open Source Software (OSS).

CoN Link: https://portal.netcom.army.mil/apps/networthiness/_layouts/NetcomCON/rqstcon2.aspx?requestId=34019

Restrictions:

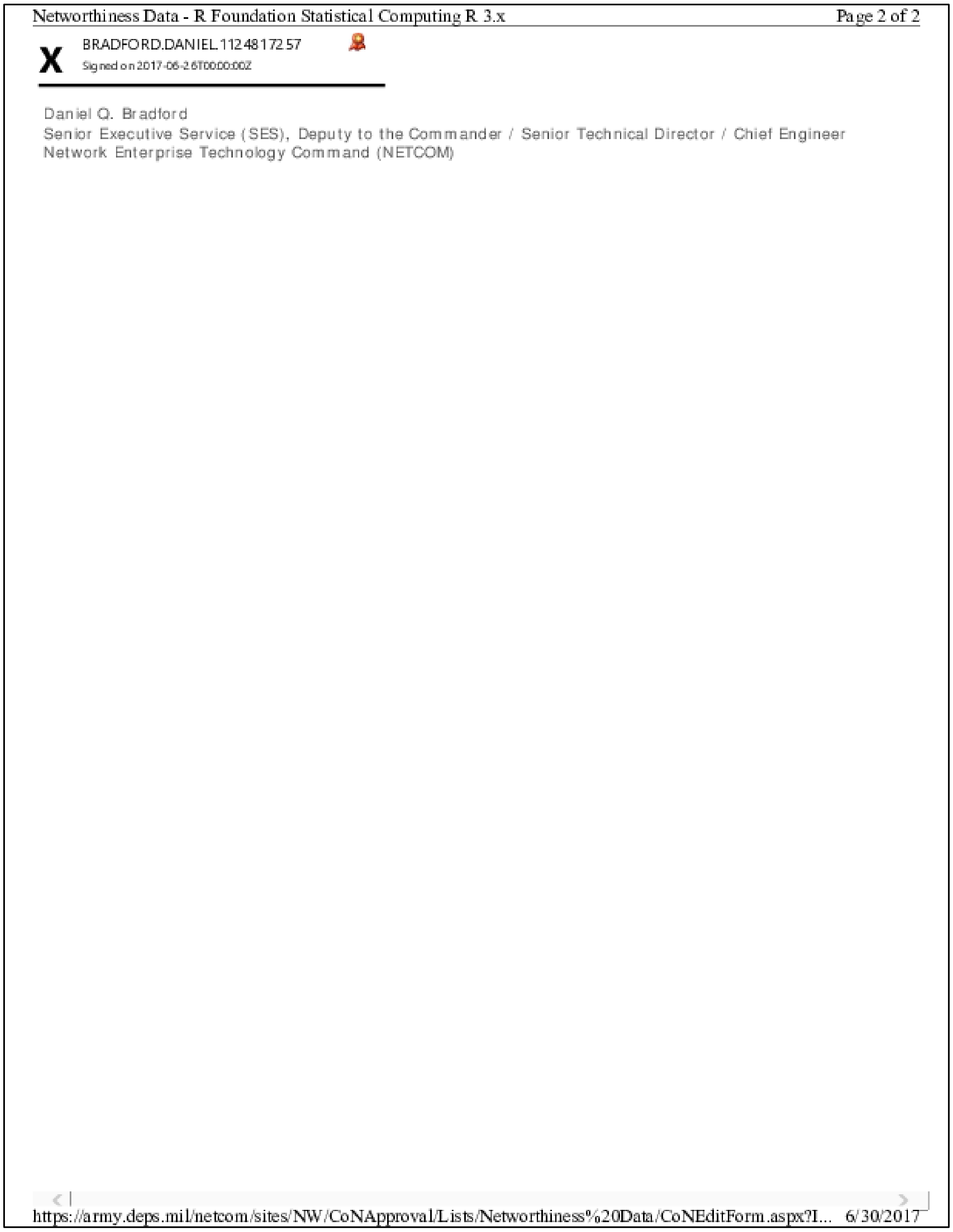
1. The organization implementing this open source software will ensure the source code is available for examination; applicable configuration implementation guidance is available, a protection profile is in place, or a risk and vulnerability assessment has been conducted with mitigation strategies implemented. This capability requires CCB approval and documentation prior to implementation. Notification of the supporting Regional Cyber Center (RCC) of local software use approval is required.

POC: netcom.hq.networthiness@mail.mil /(520)538-1199

This Certificate of Networthiness is based on a trusted standardized platform, configured per FDCC. All patches and updates will be provided by the host network administrator (NEC or DECC) in accordance with all technical directives, mandates, and IAVM. This Certificate of Networthiness is no longer valid should the assessed configuration be significantly altered. Per Army Cyber Command guidance, all operating environment will implement HBSS capabilities. Upon expiration of this Certificate of Networthiness, this application must be reassessed to ensure it is still compliant with the GNEC architecture and is still networthy. Should this software version be upgraded prior to this expiration, the new version must be evaluated for networthiness.

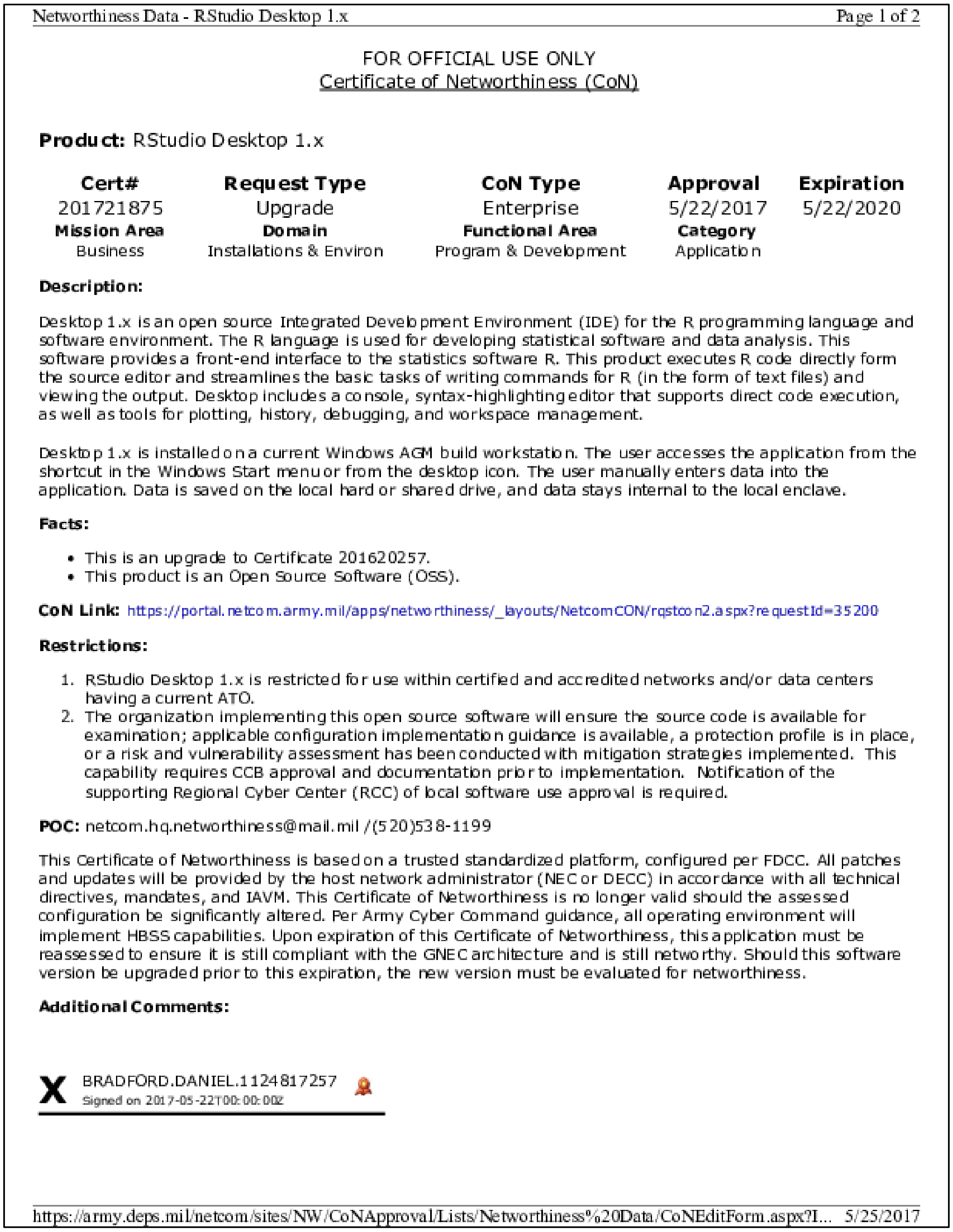
Additional Comments:

https://army.deps.mil/netcom/sites/NW/CoNApproval/Lists/Networthiness%20Data/CoNEditForm.aspx?I... 6/30/2017



X BRADFORD.DANIEL.1124817257
Signed on 2017-06-26T00:00:00Z

Daniel Q. Bradford
Senior Executive Service (SES), Deputy to the Commander / Senior Technical Director / Chief Engineer
Network Enterprise Technology Command (NETCOM)



Networthiness Data - RStudio Desktop 1.x Page 1 of 2

FOR OFFICIAL USE ONLY

Certificate of Networthiness (CoN)

Product: RStudio Desktop 1.x

Cert#	**Request Type**	**CoN Type**	**Approval**	**Expiration**
201721875	Upgrade	Enterprise	5/22/2017	5/22/2020
Mission Area	**Domain**	**Functional Area**	**Category**	
Business	Installations & Environ	Program & Development	Application	

Description:

Desktop 1.x is an open source Integrated Development Environment (IDE) for the R programming language and software environment. The R language is used for developing statistical software and data analysis. This software provides a front-end interface to the statistics software R. This product executes R code directly form the source editor and streamlines the basic tasks of writing commands for R (in the form of text files) and viewing the output. Desktop includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging, and workspace management.

Desktop 1.x is installed on a current Windows AGM build workstation. The user accesses the application from the shortcut in the Windows Start menu or from the desktop icon. The user manually enters data into the application. Data is saved on the local hard or shared drive, and data stays internal to the local enclave.

Facts:

- This is an upgrade to Certificate 201620257.
- This product is an Open Source Software (OSS).

CoN Link: https://portal.netcom.army.mil/apps/networthiness/_layouts/NetcomCON/rqstcon2.aspx?requestId=35200

Restrictions:

1. RStudio Desktop 1.x is restricted for use within certified and accredited networks and/or data centers having a current ATO.
2. The organization implementing this open source software will ensure the source code is available for examination; applicable configuration implementation guidance is available, a protection profile is in place, or a risk and vulnerability assessment has been conducted with mitigation strategies implemented. This capability requires CCB approval and documentation prior to implementation. Notification of the supporting Regional Cyber Center (RCC) of local software use approval is required.

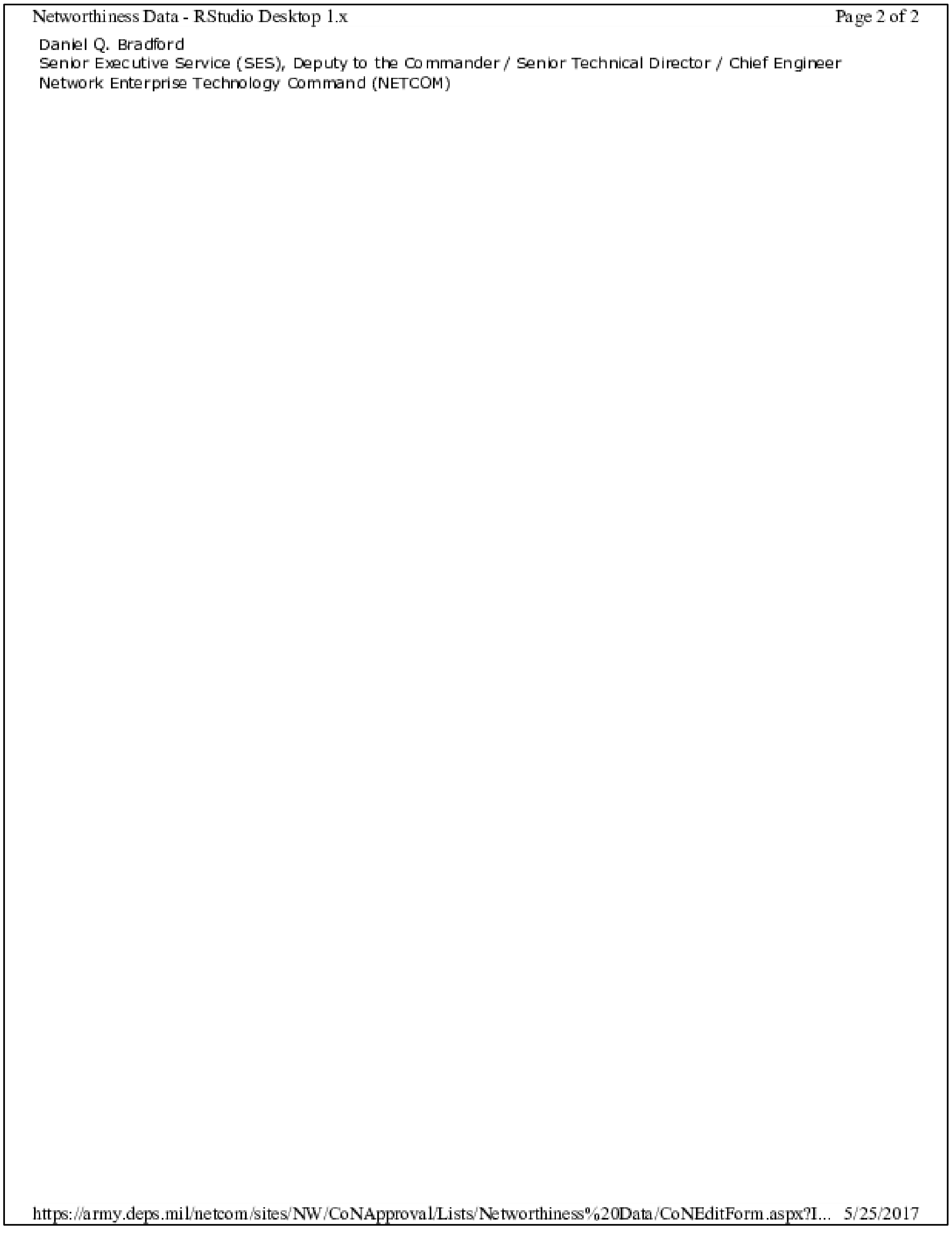
POC: netcom.hq.networthiness@mail.mil /(520)538-1199

This Certificate of Networthiness is based on a trusted standardized platform, configured per FDCC. All patches and updates will be provided by the host network administrator (NEC or DECC) in accordance with all technical directives, mandates, and IAVM. This Certificate of Networthiness is no longer valid should the assessed configuration be significantly altered. Per Army Cyber Command guidance, all operating environment will implement HBSS capabilities. Upon expiration of this Certificate of Networthiness, this application must be reassessed to ensure it is still compliant with the GNEC architecture and is still networthy. Should this software version be upgraded prior to this expiration, the new version must be evaluated for networthiness.

Additional Comments:

X BRADFORD.DANIEL.1124817257
Signed on 2017-05-22T00:00:00Z

https://army.deps.mil/netcom/sites/NW/CoNApproval/Lists/Networthiness%20Data/CoNEditForm.aspx?I... 5/25/2017



Daniel Q. Bradford
Senior Executive Service (SES), Deputy to the Commander / Senior Technical Director / Chief Engineer
Network Enterprise Technology Command (NETCOM)